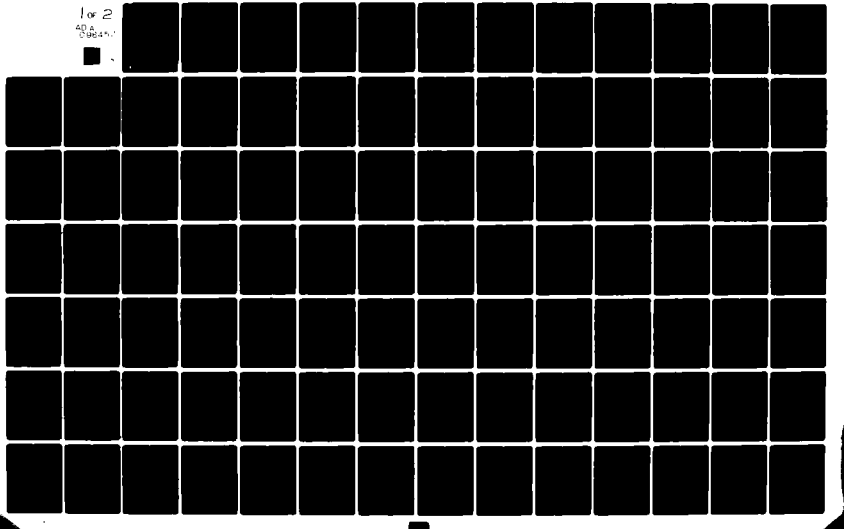
AD-A096 452 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
AN EXPERIMENTAL INVESTIGATION OF COMPUTER PROGRAM DEVELOPMENT A--ETC(U)
DEC 79 R W REITER AFOSR-77-3181
UNCLASSIFIED TR-853 AFOSR-TR-81-0214 NL

1 of 2
AD A096452



AFOSR-TR- 81-0214

LEVEL II

12

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

LEVEL II

12

Technical Report TR-853 December 1979

An Experimental Investigation of Computer Program Development Approaches and Computer Programming Metrics*

by

Robert William Reiter, Jr.

DTIC

Dissertation submitted to the Faculty of the Graduate School of the University of Maryland in partial fulfillment of the requirements for the degree of Doctor of Philosophy 1979

*Research supported in part by the Air Force Office of Scientific Research Grant AFOSR-77-3181. Computer time supported in part through the facilities of the Computer Science Center of the University of Maryland.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR-81-0214	2. GOVT ACCESSION NO.: AD A096452
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): AN EXPERIMENTAL INVESTIGATION OF COMPUTER PROGRAM DEVELOPMENT APPROACHES AND COMPUTER PROGRAMMING METRICS.	5. TYPE OF REPORT & PERIOD COVERED: Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Robert William Reiter, Jr.	8. CONTRACT OR GRANT NUMBER(s): AFOSR-77-3181
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Maryland, Department of Mathematics, College Park, Md. 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, Washington, DC 20332	12. REPORT DATE: December 1979
	13. NUMBER OF PAGES: 150
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

a large

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

There is a need in the emerging field of software engineering for empirical study of software development approaches and software metrics. An experiment has been conducted to compare three programming enviroments individual programming under an ad hoc approach, team programming under an ad hoc approach, and team programming under a disciplined methodology. This disciplined methodology integrates the use of top-down design, process design language, structured programming, code reading, and chief programmer team organization. Data was obtained for a large number of automatable software

DD FORM 1 JAN 73 1473

409022

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

metrics characterizing the software development process and the developed software product. The results reveal several statistically significant differences among the programming enviroments on the basis of the metrics. These results are interpreted as demonstrating the advantages of disciplined team programming in reducing software development costs relative to ad hoc approaches and improving software product quality relative to undisciplined team programming.

ABSTRACT

Title of Dissertation: An Experimental Investigation of Computer Program Development Approaches and Computer Programming Metrics

Robert William Reiter, Jr., Doctor of Philosophy, 1979

Dissertation directed by: Dr. Victor R. Basili
Associate Professor
Department of Computer Science

There is a need in the emerging field of software engineering for empirical study of software development approaches and software metrics. An experiment has been conducted to compare three programming environments: individual programming under an ad hoc approach, team programming under an ad hoc approach, and team programming under a disciplined methodology. This disciplined methodology integrates the use of top-down design, process design language, structured programming, code reading, and chief programmer team organization. Data was obtained for a large number of automatable software metrics characterizing the software development process and the developed software product. The results reveal several statistically significant differences among the programming environments on the basis of the metrics. These results are interpreted as demonstrating the advantages of disciplined team programming in reducing software development costs relative to ad hoc approaches and improving software product quality relative to undisciplined team programming.

DEDICATION

In honor of my mother and father,

Mary Edith Reiter

and

Robert William Reiter, Sr.

ACKNOWLEDGMENTS

This work was supported in part by the Air Force Office of Scientific Research through grant AFOSR-77-3181A to the University of Maryland. Computer time was provided in part through the facilities of the Computer Science Center of the University of Maryland.

This work could not have been accomplished without the cooperation and assistance of others. To students who participated in the experiment, colleagues who offered helpful suggestions, and faculty who reviewed the work critically, I am most grateful. Drs. Richard G. Hamlet and Ben A. Shneiderman critiqued this manuscript thoroughly on the basis of "programming sense," experimental procedure/terminology, and writing style. Drs. Marvin V. Zelkowitz and John D. Gannon imparted a healthy sense of reality and provided an appropriate measure of stimulation/inspiration throughout their lengthy service as members of my study committee.

I am indebted beyond measure, however, to two people whose professional contribution and personal sacrifice have continually enriched my work as well as my life. I thank my advisor, Dr. Victor R. Basili, for his expert guidance and patient encouragement. I thank my wife, Lowrie Ebbert Reiter, for her unselfish support and unfailing love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

1. INTRODUCTION AND OVERVIEW

In the evolution of a systematic body of knowledge, there are generally three phases of validation. The first phase is the logical development of the theory based on a set of sound principles. This is followed by the application of the theory and the gathering of evidence that the theory is applicable in practice. This usually involves some qualitative assessment in the form of case studies. The final phase is the empirical and experimental analysis of the applied theory in order to further understand its effects and better demonstrate its advantages in a controlled manner. This usually requires quantitative measurement of the relevant phenomena.

Much has been written about methodologies for developing computer software [Wirth 71; Dahl, Dijkstra & Hoare 72; Jackson 75; Myers 75; Linger, Mills & Witt 79]. Most of these methodologies are based on sound logical principles. Case studies have been conducted to demonstrate their effectiveness [Baker 75; Basili & Turner 75]. Their adoption within production ("real-world") environments has generally been successful. Having practiced adaptations of these methodologies, software designers and programmers have asserted that they got the job done faster, made fewer errors, or produced a better product. Unfortunately, solid quantitative evidence that comparatively assesses any particular methodology is scarce [Shneiderman et al. 77; Myers 78]. This is due partially to the cost and impracticality of a valid experimental setup within a production environment.

Thus the question remains, are measurable benefits derived from programming methodologies, with respect to

either the software development process or the developed software product? Even if the benefits are real, it is not clear that they can be quantified and effectively monitored. Software development is still too artistic, in the aesthetic or spontaneous sense. In order to understand it more fully, manage it more effectively, and adapt it to particular applications or situations, software development must become more scientific, in the engineering and calculated sense. More empirical study, data collection, and experimental analysis are required to achieve this goal.

This dissertation strives to contribute to software engineering research in this vital third phase of validation. The dissertation reports on an original research project dealing with three "dimensions" of software engineering:

Software development approaches, i.e., programming methodologies and environments for developing software;

Software metrics, i.e., quantifiable aspects of programming and measurements of software characteristics; and

Empirical/experimental study, i.e., the collection and statistical analysis of empirical data about software phenomena, including controlled psychological experimentation.

The immediate goals of the project were

(a) to investigate the effect of certain programming methodologies and environments upon software development phenomena,

(b) to investigate the behavior of certain quantifiable programming aspects and software measurements under different approaches to software development, and

(c) to devise and apply an investigative methodology, founded on established principles of experimental

research, but tailored for application to software engineering.

The project employed the investigative methodology to conduct and analyze a controlled experiment with software development approaches as independent variables and software metrics as dependent variables. In this way, both the effect of the software development approaches and the behavior of the software metrics were investigated scientifically.

In regard to software development approaches, the project focused on three distinct approaches, or programming environments: single programmers using an ad hoc approach, programming teams using an ad hoc approach, and programming teams using a disciplined methodology. These approaches may be characterized according to two human-factors issues: the size of the programming "team" deployed and the degree of methodological discipline employed.

In terms of team size, individual programmers working alone were compared to teams of three programmers working together. In terms of methodological discipline, an ad hoc approach allowing programmers to develop software without externally imposed methodological constraints was compared to a disciplined methodology obliging programmers to follow certain modern programming practices and procedures. This disciplined methodology consisted of an integrated set of software development techniques and team organizations including top-down design, process design language, structured programming, code reading, and chief programmer teams.

It should be noted that the terms 'methodology' and 'methodological' (in reference to software development) are used to connote an integrated set of development techniques

as well as team organizations, rather than a particular technique or organization in isolation. Part of the philosophy behind the project is the belief that, while particular techniques or organizations may generate marginal benefits individually, only a comprehensive ensemble can ensure significant gains in software development productivity and reliability.

In regard to software metrics, the project focused on the direct quantification of software development phenomena via a host of nearly two hundred programming aspects and measurements. Attention was consciously restricted to metrics exhibiting certain desirable characteristics; all of the software metrics examined in the study are quantitative (on at least an interval scale [Stevens 46]), objective (free from inaccuracy due to human subjectivity), unobtrusive (to those developing the software), and automatable (not dependent on human agency for computation).

This large set of programming aspects may be dichotimized on the basis of other criteria. Some of the aspects pertain to the software development process; others, to the developed software product. For example, the number of times that source code modules are compiled during the development period is a process measure, while the number of IF statements in the delivered program source code is a product measure. Some of the aspects are rudimentary, in that they pertain to very simple surface features or lack theoretical models to motivate intuitive appeal; others are elaborative, in that they aim at more complicated underlying features or possess provocative theoretical models. For example, the measurements mentioned above are both rudimentary, while the program changes metric [Dunsmore & Gannon 77] and the cyclomatic complexity metric [McCabe 76] are elaborative.

CHAPTER I

In regard to empirical/experimental study, the project combined both empirical data collection and controlled psychological experimentation in a laboratory-like setting.

The project involved extensive observation of forty-five programmers developing working software systems, averaging twelve hundred lines of code each, from scratch during a five week period. These programmers were divided into three disjoint groups of "teams," each following one of the three software development approaches mentioned above. Multiple replications of a specific software development task were performed independently and concurrently within each group under conditions as otherwise identical as possible.

In addition to some subjective qualitative observation via questionnaires, interviews, etc., objective quantitative observation was achieved by automatically and unobtrusively monitoring the computer activites of the programming "teams." For each replication, successive versions of the software being developed by that "team" were captured in an historical data bank that recorded details of the development process and product. Raw scores for the software metrics mentioned above were extracted from the data bank and summarized via simple descriptive statistics. Specifically, the mean values and standard deviations observed within each group on the various quantifiable programming aspects constitute the immediate results of the project as an empirical data collection effort.

The project followed a preplannd experimental design in which extraneous factors were held constant wherever possible, to insure that differences in the software metrics would be attributable to the different software development approaches. The metrics' raw scores were analyzed using

nonparametric inferential statistics to obtain an objective conclusion for each measured aspect. As precise statements of the statistically significant differences observed among the three programming environments on the basis of the measured aspects, these objective conclusions constitute the immediate results of the project as a controlled experiment. By testing for differences in either the location (expected value) or the dispersion (variability) of the software metrics, the experiment addressed both the expectancy and predictability of software development phenomena.

The experiment combined elements of both confirmatory and exploratory data analysis. Some so-called confirmatory programming aspects had been earmarked as promising indicators of important software characteristics in advance of conducting the experiment. Hypotheses had been formulated, on the basis of the programming environments' suspected effects, regarding the expected objective conclusions for these confirmatory aspects. The project included other so-called exploratory programming aspects in order to investigate the software development process and conduct more thoroughly.

The project was concerned with investigating an entire software development project of nontrivial size in a quasi-realistic setting. The experiment was conducted within an academic environment in a laboratory or proving-ground fashion so that an adequate experimental design could be achieved while simulating a production environment. In this way, the project reached a reasonable compromise between "toy" experiments, which facilitate elaborate experimental designs but often suffer from artificiality, and "production" experiments, which offer industrial realism but incur prohibitively high costs.

The project's basic premise was that distinctions among these programming environments exist both in the process and in the product. With respect to the developed software product, the disciplined team should approximate the individual programmer or at least lie somewhere between the individual programmer and the ad hoc team, with regard to product characteristics (such as number of decisions coded and global data accessibility). This is because the disciplined methodology should help the team act as a mentally cohesive unit during the design, coding, and testing phases. With respect to the software development process, the disciplined team should have advantages over both individuals and ad hoc teams, displaying superior performance on cost-related factors such as computer usage and number of errors made. This is because of the discipline itself and because of the ability to use team members as resources for validation.

The study's findings revealed several programming characteristics for which statistically significant differences do exist among the groups. The disciplined teams used fewer computer runs and apparently made fewer errors during software development than either the individual programmers or the ad hoc teams. The individual programmers and the disciplined teams both produced software with essentially the same number of decision statements, but software produced by the ad hoc teams contained greater numbers of decision statements. For no characteristic was it concluded that the disciplined methodology impaired the effectiveness of a programming team or diminished the quality of the software product.

The remainder of this dissertation is a comprehensive report on the software engineering research project introduced above. Chapter II reviews appropriate background

and related research from published literature. Chapter III recounts specific details of the experiment itself. Chapter IV briefly describes all of the programming aspects and measurements, while Chapter V discusses the elaborative ones in depth. Chapter VI depicts the investigative methodology used to plan, execute, and analyze the experiment. Chapters VII and VIII present the experiment's results, segregated into objective findings and interpretative discussion, respectively. Chapter IX summarizes the completed project, draws general conclusions regarding its contribution to software engineering, and mentions possible directions for continued research in this area.

CHAPTER II

II. BACKGROUND AND RELATED RESEARCH

This chapter reviews the general background for this research project and surveys related work published in the open literature. For each of the three "dimensions" of software engineering outlined in Chapter I, specific instances of research in that area will be mentioned and loosely characterized, in order to show appropriate similarities and constrasts with this work. As a catalog of related research, the chapter is intended to be merely representative, not exhaustive.

Software Development Approaches

There has been considerable concern regarding programming methodologies over the past decade since the advent of structured programming and the dawning of software cost consciousness. Software "practitioners" (i.e., programmers, designers, systems analysts, and managers) have sought better ways to channel their energies toward producing cost-effective, reliable software. Although a broad spectrum of concerns--spanning all phases of the software life-cycle and covering the full range of system size and performance constraint--could be considered here, attention has been restricted to methodology for programming-in-the-small*: designing, implementing, and testing computer programs to solve problems small enough to be well-understood by a suitably trained individual. In other words, the focus is on approaches for the kind of software development that typical programmers/analysts in typical software shops are accustomed to doing.

---

* As used here (and below), the meanings of the terms 'programming-in-the-small' and 'programming-in-the-large' are clear from the context, but they differ slightly from the meanings popularized by Dr. H.D. Mills.

CHAPTER II

A number of good ideas on how to develop software, covering techniques for how to proceed as well as organizations for managing people and communicating information, have been (or are being) devised, demonstrated, perfected, and accepted into everday practice. Popular examples include the following:

structured programming [Dahl, Dijkstra & Hoare 72; Mills 72; Basili & Baker 77; Linger, Mills & Witt 79],
stepwise refinement [Wirth 71],
chief programmer teams [Baker 72; Baker 75; Brooks 75],
process design language (PDL) [Linger, Mills & Witt 79],
top-down design,
functional expansion,
design/code reading and walk-throughs [Fagan 76],
data abstraction/encapsulation and information hiding,
iterative enhancement [Basili & Turner 75; Turner 76],
the Michael Jackson method [Jackson 75; Hughes 79], and
composite design [Myers 75].

These approaches and their highly touted benefits have been the subject of much written promotion and verbal discussion. Indeed, several can boast of mathematical foundations or formal explication to support their underlying principles or mechanisms; for others, there are extensive tutorials on how to apply them in practical situations; and some have been embodied in programming languages or packaged into automated tools. All of this attention, plus the favorable experiences of software practitioners, seems to indicate that these software development approaches do succeed in improving the efficiency of the development process or the quality of the developed product to some degree.

but there is little empirical evidence to confirm the advantages of these approaches or measure their benefits. In several instances, case studies have been performed, often in a pioneering spirit, to demonstrate particular approaches; these case studies have usually involved qualitative assessment, with only limited or uncontrolled forms of quantitative assessment. Comparative assessment of software development approaches is even rarer: only a few controlled experiments [Shneiderman et al. 77; Myers 78] have been conducted, and they have generally focused on the use of particular techniques in isolation. The difficulty of investigating the effects of software development approaches stems precisely from the fact that they pertain to the least understood and most expensive elements in software engineering: human beings.

Software Metrics

There has been considerable interest in software metrics over the past half decade in response to a growing realization of how "invisible," imponderable, and uncontrollable software can be. Software "scientists" have been seeking ways to measure software phenomena. Broadly interpreted, their efforts may be characterized as attempting to quantify process efficiency and product quality.* The software measurement domain extends from the concrete details of a program, including its fine structure and the resource expenditure required to produce it, to its abstract characteristics: reliability, cost-effectiveness,

---

* This concept of product quality is meant to include instantaneous, as well as evolutionary, considerations. The former considerations pertain to both static (at compile time) and dynamic (at execution time) features of a program, as it exists at a given point along its life-cycle. The latter considerations pertain to issues of software maintenance and software management throughout the life-cycle. The software measures in this dissertation address product quality only in its instantaneous, static sense.

complexity, modularity, comprehensibity, modifiability, etc.

Because measurement is essential to most forms of engineering, software metrics rightfully deserve a central place within the emerging discipline of software engineering. As in other technologies, the underlying assumption is that appropriate measurement is the key to effective control. It has been demonstrated [Gilb 77] that the general concept of software measurement can be applied to a variety of programming issues: many interesting suggestions were made regarding how and why to measure software. But the metrics discussed by Gilb are vaguely defined and superficial. The problem is that meaningful measurement of software is extremely difficult, because of software's intricate structure of concrete detail and because of the tenuous relationship between its concrete details and abstract characteristics. An additional problem is the lack of well-understood and commonly accepted terminology to describe the software phenomena to be measured.

However, a number of well-defined and fairly credible software metrics have been proposed and evaluated, usually in conjunction with a motivating model or some intuitional underpinnings. The program changes metric [Dunsmore & Gannon 77; Dunsmore 78] extracts an error count algorithmically from the textual revisions made to source code during program development. The cyclomatic complexity metric [McCabe 76] counts the number of "basic" control-flow paths in a program. The data bindings metric [Stevens, Myers & Constantine 74; Basili & Turner 75; Turner 76] counts communication paths between code segments via data variables. The various metrics from software science theory [Halstead 77]--program length, program volume, language level, effort, etc.--provide a unified system of

measurements for the size of a program, the amount of information it contains, the level of abstraction it expresses, the amount of mental effort required to produce or comprehend it, etc. The error-day metric [Mills 76] is an index of how early errors are detected and corrected during software development. The span metric [Elshoff 76b] is an index of the extent to which a program's data variables remain "live" (i.e., continue to affect control flow and data value determination).

Each metric mentioned above has been examined empirically to one degree or another; but few software metrics have been investigated in controlled experiments, and there is little research comparing metrics or examining their interrelationships empirically. Further elaboration and discussion of individual software metrics is deferred to Chapters IV and V since many were examined in this reseach project.

Empirical/Experimental Study

First-hand observation of software phenomena in the "wild," so to speak, has long been regarded as a unique source of information and the ultimate form of validation. Ever since Knuth rummaged through wastebaskets at computer centers for discarded listings of Fortran programs [Knuth 71], software "technicians" have been interested in watching software be developed, to see how the latest intuitive opinions or theoretical models fare against reality. Ideally, it is useful to distinguish between data collection efforts (with descriptive statistical analyses) and controlled experimentation efforts (with inferential statistical analyses); but, in practice, elements of both are sometimes combined within the same empirical study.

CHAPTER II

Generally speaking, the purpose of data collection efforts has been to examine the behavior of software metrics and models under realistic conditions. A number of data collection efforts have been aimed at progamming-in-the-large,* focusing on models of gross behavior (i.e., cost, productivity, resource estimation) during large- to medium-scale software development. At IBM [Walston & Felix 77] data was collected via project reporting forms in order to measure productivity on production software developments. At NASA/Goddard [Basili et al. 77] data is being collected via information forms in order to evaluate cost or resource estimation models and to study software error phenomena.

Other data collection efforts, focusing on small- to medium-scale software development, have been aimed at quantitatively characterizing software's fine structure. In studies at GM [Elshoff 76b; Elshoff 76a], a large set of commercial PL/1 programs was collected and measured according to a host of quantifiable programming aspects and software metrics, including the span metric and the software science metrics.

Generally speaking, the purpose of controlled experimentation efforts has been to evaluate the effects of programming language features, human factors issues, and programming methodologies upon software phenomena and abstract characteristics. Usually, the language features experiments are done from a computer scientist's viewpoint, while the human factors experiments are done from a psychologist's viewpoint. However, because of areas of natural overlap between these two concerns, some experiments fall into both categories. Together they comprise the bulk of controlled experimentation in software

---

* See earlier footnote.

engineering.

There are several well-known examples of controlled experimentation on programming language features. Weissman [Weissman 74a; Weissman 74b] conducted experiments on how programming features affect the psychological complexity of software; the features included commenting, indentation, mnemonic variable names, and control structures. Gannon [Gannon 75; Gannon & Horning 75] conducted an experiment on how programming language features affect software reliability and the presence/persistence of errors; the features included statement vs. expression orientation, data variable scope conventions, and expression evaluation order. Later, Gannon [Gannon 77] ran experiments to examine how data typing conventions affect software reliability. Using the same empirical data, Dunsmore [Dunsmore & Gannon 77; Dunsmore 78] examined how programming "complexity" is affected by programmer-controllable variations in programming features. "Complexity" was measured algorithmically by the program changes metric; the features included statement nesting depth, frequency of data references, and data communication mechanism preference.

There are several well-known examples of controlled experimentation on human factors issues. Several experiments [Sime, Green & Guest 73; Green 77] have been conducted on the comprehensibility of different mechanisms for implementing conditional branching. Several experiments [Sheppard et al. 79] have been conducted on the effect of modern coding practices, such as structured coding, mnemonic variable names, and style of commenting upon the ease of performing comprehension, modification, and debugging tasks.

Finally, there are a few well-known examples of controlled experimentation on programming methodologies.

CHAPTER II

Several experiments were conducted [Shneiderman et al. 77] to evaluate the utility of detailed flowcharting (as a design tool and documentation aid) in program composition, comprehension, debugging, and modification tasks; novice progamming students were employed as subjects, with short (i.e., less than 150 lines) Fortran programs as test materials. Some experiments were also conducted [Myers 78] to evaluate the utility of code reading and walkthroughs in debugging tasks; experienced professional programmers were employed as subjects, with a short PL/1 program as test material. To date, however, controlled experimentation on programming methodologies has been limited in scope. Experimental studies have not involved programming activities spanning multiple phases of the software life-cycle and requiring the natural integration of multiple programming tasks. Nor have experimental studies used nontrivial test materials requiring sustained effort lasting several weeks and involving several hundred lines of code.

CHAPTER III

III. INVESTIGATION SPECIFICS

This chapter outlines the surroundings in which the experiment was conducted, the experimental design that was employed, the programming methodologies that were compared, the data collection and reduction that was performed, and the programming aspects that were measured.

Surroundings

Several circumstances surrounding the experiment contribute significantly to the context in which its results must be appraised. These include the setting in which the experiment was conducted, the people who participated as subjects, the software development project that served as the experimental task, the computer programming language in which the software was written, and the computer system and access mode that were used during development.

The experiment was conducted during the Spring 1976 semester, January through May, within regular academic courses given by the Department of Computer Science on the College Park campus of the University of Maryland. Two comparable advanced elective courses were utilized, each with the same academic prerequisites. The experimental task and treatments were built into the course material and assignments. Everyone in the two classes participated in the experiment; they cooperated willingly and were aware of being monitored, but had no knowledge of what was being observed or why.

The participants were advanced undergraduate and graduate students in the Department of Computer Science. On the whole, they were reasonably competent computer

programmers, all having completed at least four semesters of programming course work and some having as much as three years' professional programming experience in government or industry. Generally speaking, they were familiar with the implementation language and the host computer system, but inexperienced in team programming and the disciplined methodology.

The programming application was a simple compiler, involving string processing and translation (via scanning, parsing, code generation, and symbol table management) from an Algol-like language to zero-address code for a hypothetical stack machine. The total task was to design, implement, test, and debug the complete computer software system from given specifications. The scope of the project excluded both extensive error handling and user documentation. The project was of modest but nonnegligible difficulty, requiring between one and two man-months of effort. The size of the resulting systems averaged over 1200 lines of high-level language source code. All facets of the project itself were fixed and uniform across all development "teams." Given the same specifications, computer resource allocation, calendar time allotment, host machine, implementation language, debugging tools, etc., each "team" worked independently to build its own system. The delivered systems each ran (i.e., they worked) and passed an independent acceptance test.

The implementation language was the high-level, structured-programming language SIMPL-T [Basili & Turner 76]. This language was designed and developed at the University of Maryland where it is taught and used extensively in regular Department of Computer Science courses. SIMPL-T contains the following control constructs: sequence, ifthen, ifthenelse, whiledo, case, exit from loop,

and return from routine (but no goto). SIMPL-T allows essentially three levels of data declaration scope (i.e., local to an individual routine, global across the several routines of an individual module, or entry-global across the routines of several modules), but routines may not be nested. Adhering to a philosophy of "strong typing," the language supports integer, character, and string data types and single dimension array data structures. It provides the programmer with automatic recursion and PL/1-like string-processing capabilities. (Additional details regarding the SIMPL-T programming language are interspersed among the explanatory notes in Chapter IV.)

The host computer system was the campus-wide computing facility, a Univac 1100 machine with the usual Exec 8 operating system. This system supports, in its fashion, both batch access (via punch cards) and interactive time-sharing* access (via TTY or CRT terminals). The participants were well acquainted with the system and accustomed to either access mode. During the experiment, the participants were allowed to choose whichever access mode they preferred and could switch freely between modes. Almost everyone consistently preferred the interactive access mode; only one person--in the AI group (see below), by the way--used the batch access mode extensively.

Experimental Design

The major elements of an experimental design are its units, treatment factors, treatment factor levels, observed variables, local control, and management of extraneous factors. (Cf. [Ostle and Mensing 75, chap. 9] for a general treatment of these elements.)

---

* Called "demand" in Univac terminology.

CHAPTER III

An experimental unit is that object to which a single treatment is applied in one replication of the event known as the "basic experiment." In this study, the "basic experiment" was the accomplishment of a specific software development project (see above), and the experimental unit was the software development team (i.e., a small group of people working together to develop the software). A total of 19 replications of this "basic experiment," each performed concurrently and independently by a separate experimental unit, were involved in this experiment.

Most experiments are concerned with on one or more independent variables and the behavior of a one or more dependent variables as the independent variables are permitted to vary. These independent variables are known as experimental treatment factors. This experiment focused on the approach used to develop software, as the single experimental treatment factor.

Experiments usually involve some deliberate variation in the experimental treatment factor(s). Different values or classifications of the factor(s) are known as the experimental treatment factor levels. In this experiment, three levels were selected for the software development approach factor. Conceived as variations in two human-factors-in-programming issues, size of development "team" and degree of methodological discipline, the experimental treatment factor levels are denoted by the following mnemonics:

AI -- individual programmers working alone, following an ad hoc approach (see below);

AT -- teams of three programmers working together, following an ad hoc approach (see below); and

DT -- teams of three programmers working together, following a disciplined methodology (see below).

CHAPTER III

During an experiment, observations of the dependent variable(s) are made for each experimental unit. An experiment's immediate objective is to ascertain the relationship between the experimental treatment factor levels and the experimental observed variables. In this experiment, the observed variables were quantifiable programming aspects, or metrics (see below), of the software development process or the developed software product. A large set of such aspects were considered in the study. Technically speaking, this amounted to conducting a series of simultaneous univariate experiments, one for each programming aspect, all sharing a common experimental design and all based on the same empirical data sample.

Experimental local control addresses the configuration by which (a) experimental units are obtained, (b) units are placed into groups, and (c) groups are subjected to different experimental treatments (i.e., specific combinations of experimental treatment factor levels). Local control is employed in the design of an experiment in order to increase its statistical efficiency or to improve the sensitivity/power of statistical test procedures. Experimental local control usually incorporates some form of randomization--a basic principle of experimental design--since it is necessary for the validity of statistical test procedures.

For this experiment, subjects were obtained on the basis of course enrollment: since the experiment was embedded within two academic courses, every student enrolled in those courses automatically participated in the experiment. Software development "teams" were formed among these subjects. In the one course, the students were allowed to choose between segregating themselves as individual programmers or combining with two other

classmates as three-person programming teams. In the other course, the students were assigned (by the researcher) into three-person teams. The two academic courses themselves provided the variation in methodological discipline. The atmosphere of the first course was conducive to an ad hoc approach to programming, while the disciplined methodology was stressed in the second course. In this manner, three experimental treatments (corresponding to the three experimental treatment factor levels AI, AT, and DT) were created, and three groups of 6, 6, and 7 units (respectively) were exposed to them.

There are usually several extraneous factors, other than the ones identified as experimental treatment factors, that could influence the behavior being observed in an experiment. Many experiments (includings this one) follow a reductionist paradigm, which seeks to control for all variables except a select few, so that the effect of the independent variables upon the dependent variables can be isolated and measured. In this experiment, a variety of programming factors which do affect software development were given conscious consideration as extraneous variables:

- programming application and/or project
- project specifications
- implementation language
- calendar schedule
- available computer resources
- available automated tools

Wherever possible, these variables were held constant by explicitly treating all experimental units in the same manner.

Unfortunately, the ideal reductionist paradigm can only be approximated, because of factors which are suspected of strong influence on the behavior of interest, but which

cannot be explicitly controlled within the experimental design. In this experiment, there were two such factors: the personal ability/experience of the participants and the amount of actual time/effort they (as students with other classes and responsibilities) chose to devote to the project. However, information from a pretest questionnaire was used to balance the personal ability/experience of the participants in the disciplined teams (only), by first partitioning the group DT students into three equal-sized categories (high, medium, low) based on their grades in previous computer courses and their extracurricular programming experience, and then randomly selecting one student from each category to form each team.

For the statistical model employed to analyze this experiment, it was necessary to assume homogeneity among the participants with respect to personal factors such as ability and/or experience, motivation, time and/or effort devoted to the project, etc. As a reasonable measure of individual programmer skill levels under the circumstances of this study, the participants' grades from a particularly pertinent prerequisite course provided a post-experimental confirmation of at least one facet of this assumed homogeneity: the distribution of these grades among the three experimental groups would have displayed the same degree of homogeneity as was actually observed in over 9 out of 10 purely random assignments of the participants to the groups. If anything, in the researcher's opinion, the participants in group AI seemed to have a slight edge over those in groups AT and DT with respect to native programming ability, while groups AI and AT seemed slightly favored over group DT with respect to formal training in the application area.

CHAPTER III

Programming Methodologies

The disciplined methodology imposed on teams in group DT consisted of an integrated set of state-of-the-art techniques, including top-down design, process design language (PDL), functional expansion, design and code reading, walk-throughs, and chief programmer team organization. These techniques and organizations were taught as an integral part of the course that the subjects were taking, using [Linger, Mills & Witt 79], [Basili & Baker 75], and [Brooks 75] as textbooks. Since the subjects were novices in the methodology, they executed it to varying degrees of thoroughness and were not always as successful as seasoned users of the methodology would be.

The disciplined methodology prescribed the use of a PDL for expressing the design of the problem solution. The design was elaborated in a top-down manner, each level representing a solution to the problem at a particular level of abstraction and specifying the functions to be expanded at the next level. The PDL consisted of a fixed set of structured control and data structures, plus an open-ended designer-defined set of operators and operands corresponding to the level of the solution and the particular application. Design and code reading involved the critical review of each team member's PDL or code by at least one other member of the team. Walk-throughs represented a more formalized presentation of an individual's work to the other members of the team in which the PDL or code was explained step by step. Under the chief programmer team organization, the chief programmer defined the top-level solution to the problem in PDL, designed and implemented key portions of code himself, and assigned subtasks to the other two team members. Each of these programmers, in turn, code-read for the chief programmer, designed or coded their assigned

subpieces, and performed librarian activities (i.e., entering or revising code stored on-line, making test runs, etc.).

Two variants of chief programmer team organization, denoted CP and M, were employed. In both cases, one member of the team (the chief programmer or the manager) was responsible for designing and refining the top-level solution to the problem in PDL, identifying system components to be implemented, and defining their interfaces. The two other team members (the programmers) were each responsible for designing or coding various system components, as assigned by the chief programmer or manager. In the CP case, the chief programmer maximized his coding duties by implementing the key code himself, and the programmers performed librarian activities (i.e., entering or revising code stored on-line, making test runs, etc.). In the M case, the manager minimized his coding duties by acting as librarian and yielding greater responsibility for implementation to the programmers. Although there were (supposedly) four CP teams and three M teams in group DT, this distinction between the CP and M variants of chief programmer team organization is not utilized in the present study, since it is believed that the impact of their common features transcends any impact due to their differences. Moreover, in actual practice, it was observed that the CP and M variants are only identifiable extrema along a continuum and that the group DT teams all gravitated toward a comfortable compromise in this respect.

Each individual or team in groups AI or AT was allowed to develop the software entirely in a manner of his or their own choosing, which is herein referred to as an ad hoc approach. No methodology was taught in the course these subjects were taking. Informal observation by the

researcher confirmed that approaches used by the individuals and ad hoc teams were indeed lacking in discipline and did not utilize the key elements of the disciplined methodology (e.g., an individual working alone cannot practice code reading, and it was evident that the ad hoc teams did not employ a PDL or a formal top-down design).

Data Collection and Reduction

Due to the partially exploratory nature of the experiment in terms of differences to be discovered in the project and process, as much information was collected as could be done in an efficient and unobtrusive manner. A variety of information sources was used. Individual questionnaires revealed the personal background and programming experience of each participant. Private team interviews and in-class team reports provided information regarding individual performance on the project. "Run logs" and computer account billing reports gave a record of the computer activity during the project. Special module compilation and program execution processors (invoked on-line via very slight changes to the regular command language) created an historical data bank of source code and test data accumulated throughout the project development.

The data bank provided the principal source of information analyzed in the current investigation and other information sources have been utilized only in an auxiliary manner (if at all). Thus, data collection for the experiments themselves was automated on-line, with essentially no interference to the programmer's normal pattern of actions during computer (terminal) sessions. The final products were isolated from the data bank and measured for various syntactic and organizational aspects of the finished product source code. Effort and cost data were

also extracted from the data bank. The inputs to the analysis, in the form of scores for the various programming aspects, reflect the quantitatively measured character of the product and the process. Much of the data reduction was done automatically within a specially instrumented compiler. Some was done manually (e.g., examining characteristics across modules). Due to the underlying collection and reduction mechanism, which was uniformally applied to all experimental units, the data used in the analysis has the characteristics of objectivity, uniformity, and quantitativeness and is measured on an interval scale of measurement [Stevens 46]. The raw scores for the measured programming aspects are summarized in Apppendix 1.

Programming Aspects and Metrics

The dependent variables studied in this experiment are called programming aspects. They represent specific isolatable and observable features of programming phenomena. Furthermore, they are measured in an objective and automatable manner (i.e., they could be extracted or computed directly on-line from information readily obtainable from operating systems and compilers). For each programming aspect there exists an associated metric, a specific algorithm which ultimately defines that aspect and by which it is measured.

The programming aspects may be categorized as either process- or product-related, on the basis of what they measure. Process aspects represent characteristics of the development process, in particular, the cost and required effort as reflected in the number of computer job steps (or runs) and the amount of textual revision of source code during development. Product aspects represent characteristics of the final product that was developed, in

particular, the syntactic content and organization of the symbolic source code. Examples of product aspects are number of lines, frequency of particular statement types, average size of data variables' scope, etc.

The programming aspects may also be categorized as either rudimentary or elaborative, on the basis of their conceptual nature. The rudimentary aspects are conceptually quite simple, reflecting ordinary surface features of the process or product. For example, the numbers of data variables and routines in a program are rudimentary aspects; they pertain to the sheer size of the software and are somewhat uninteresting in themselves. The elaborative aspects are conceptually more subtle, reflecting deeper characteristics of the process or product. For example, the number of times pairs of routines communicate via data variables (see the data bindings metric below) is an elaborative aspect; it pertains to the software's modularity and is intuitively appealing.

Finally, the programming aspects may be categorized as either confirmatory or exploratory, on the basis of the motivation for their inclusion in the study. The confirmatory aspects had been consciously planned in advance of collecting and extracting the data, because intuition suggested that they would serve well as quantitative indicators of important qualitative characteristics of sofware development phenomena. It was predicted a priori that these confirmatory aspects would verify the study's basic premises regarding the programming environments being investigated in the experiment. The exploratory aspects were considered mainly because they could be collected and extracted cheaply (even as a natural by-product sometimes) along with the confirmatory aspects. There was little serious expectation that these exploratory aspects would be

useful indicators of differences among the groups; but they were included in the study with the intent of observing as many aspects as possible on the off chance of discovering any unexpected tendency or difference. Thus, this study combines elements of both confirmatory and exploratory data analysis within one common experimental setting [Tukey 69]. The confirmatory programming aspects are identified in the accompanying tables by being flagged with asterisks; the exploratory programming aspects are unflagged.

It should be noted that a large percentage of the product aspects fall into the rudimentary-exploratory category. On the whole, these product aspects represent a fairly extensive taxonomy of the surface features of software. The idea that important software qualities (e.g., "complexity") could be measured by counting such surface features has generally been disregarded by some researchers as too simplistic (e.g., [Mills 73, p. 232]). A resolve to study these surface features empirically, to see if something might turn up, before rejecting the underlying idea, was partially responsible for their inclusion in the study.

The particular programming aspects examined in this investigation are presented in Chapters IV and V. A complete list of aspects, together with explanatory notes, is given in Chapter IV, with definitions for the nontrivial or unfamiliar metrics. Chapter V contains a in-depth discussion of the elaborative aspects.

CHAPTER IV

IV. GLOSSARY OF PROGRAMMING ASPECTS

This chapter presents all of the programming aspects examined in the study. The goal of this chapter, in conjunction with the next, is to describe each programming aspect and, where appropriate, to motivate its intuitive appeal as a software metric. Because the brief explanatory notes within this chapter do not adequately describe a certain subset of the aspects (namely, the elaborative aspects), they are further discussed within the next chapter.

Table 1 lists the programming aspects examined in this investigation. They appear grouped according to definitionally related categories, with indented qualifying phrases to specify variants of general aspects. When referring to an individual aspect, a concatenation of the major phrase with any qualifying phrases (separated by \ symbols) is used.* For example, the aspect label

COMPUTER JOB STEPS\MODULE COMPILATION\UNIQUE

refers to a metric involving computer job steps that are module compilations in which the source code is unique from all other compiled versions.

In order to avoid any misunderstanding, a redundancy issue must be stated and properly appreciated. Several instances of duplicate programming aspects exist; that is, some logically unique aspects reappear with another label, in order to provide alternative views of a given metric or to round out a group of related aspects. For example, the FUNCTION CALLS aspect and the STATEMENT TYPE COUNTS\

---

* Aspect labels are always written completely in uppercase letters, while references to general concepts appear in lowercase letters, with initial or defining occurrences underlined.

Table 1. Programming Aspects

N.B. Parenthesized numbers refer to the explanatory notes in Chapter IV. Asterisks mark the confirmatory aspects; the exploratory aspects are unmarked.

rudimentary process aspects

(1)	*	COMPUTER JOB STEPS
(2)	*	MODULE COMPILATION
(7)	*	UNIQUE
(3)		IDENTICAL
(4)	*	PROGRAM EXECUTION
(5)		MISCELLANEOUS
(6)	*	ESSENTIAL
(7)		AVERAGE UNIQUE COMPILATIONS PER MODULE
(8)		MAX. UNIQUE COMPILATIONS F.A.O. MODULE

MAX. is an abbreviation for MAXIMUM
F.A.O. is an abbreviation for FOR ANY ONE

elaborative process aspects

(9)	*	PROGRAM CHANGES

rudimentary product aspects

(10)	*	MODULES
(11)	*	SEGMENTS
(12)		SEGMENT TYPE COUNTS :
(11)		FUNCTION
(11)		PROCEDURE
(12)		SEGMENT TYPE PERCENTAGES :
(11)		FUNCTION
(11)		PROCEDURE
(13)		AVERAGE SEGMENTS PER MODULE
(14)	*	LINES
(15)	*	STATEMENTS
(16)		STATEMENT TYPE COUNTS :
(17)		:=
(18)	*	IF
(19)	*	CASE
(20)	*	WHILE
(21)	*	EXIT
(24,99)		(PROC)CALL
(23,99)		NONINTRINSIC
(23,99)		INTRINSIC
(24)	*	RETURN
(16)		STATEMENT TYPE PERCENTAGES :
(17)		:=
(18)	*	IF
(19)	*	CASE
(20)	*	WHILE

TABLE 1

(21)	*	EXIT
(22)		(PROC)CALL
(23)		NONINTRINSIC
(23)		INTRINSIC
(24)	*	RETURN
(25)	*	AVERAGE STATEMENTS PER SEGMENT
(26)	*	AVERAGE STATEMENT NESTING LEVEL
(27)	*	DECISIONS
(22,99)		FUNCTION CALLS
(23,99)		NONINTRINSIC
(23,99)		INTRINSIC
(28)	*	TOKENS
(28)	*	AVERAGE TOKENS PER STATEMENT
(29)		INVOCATIONS
(11,99)		FUNCTION
(23,99)		NONINTRINSIC
(23,99)		INTRINSIC
(11,99)		PROCEDURE
(23,99)		NONINTRINSIC
(23,99)		INTRINSIC
(23)		NONINTRINSIC
(23)		INTRINSIC
(30)		AVG. INVOCATIONS PER (CALLING) SEGMENT
(11)		FUNCTION
(23)		NONINTRINSIC
(23)		INTRINSIC
(11)		PROCEDURE
(23)		NONINTRINSIC
(23)		INTRINSIC
(23,99)		NONINTRINSIC
(23)		INTRINSIC
(31,99)		AVG. INVOCATIONS PER (CALLED) SEGMENT
(11)		FUNCTION
(11)		PROCEDURE
(32)		DATA VARIABLES
(37)		DATA VARIABLE SCOPE COUNTS :
(33)	*	GLOBAL
(34)		ENTRY
(35)		MODIFIED
(35)		UNMODIFIED
(34)		NONENTRY
(35)		MODIFIED
(35)		UNMODIFIED
(35)		MODIFIED
(35)		UNMODIFIED
(33)		NONGLOBAL
(33)	*	PARAMETER
(36)		VALUE
(36)		REFERENCE
(33)	*	LOCAL
(37)		DATA VARIABLE SCOPE PERCENTAGES :
(33)	*	GLOBAL
(34)		ENTRY
(35)		MODIFIED
(35)		UNMODIFIED
(34)		NONENTRY
(35)		MODIFIED
(35)		UNMODIFIED
(35)		MODIFIED
(35)		UNMODIFIED

TABLE 1

```
(33)       |    NONGLOBAL                                          |
(33)    *  |       PARAMETER                                       |
(36)       |          VALUE                                        |
(36)       |          REFERENCE                                    |
(33)    *  |       LOCAL                                           |
           |-------------------------------------------------------|
(34)       | AVERAGE GLOBAL VARIABLES PER MODULE                   |
(34)       |    ENTRY                                              |
(34)       |    NONENTRY                                           |
(35)       |    MODIFIED                                           |
(35)       |    UNMODIFIED                                         |
           |-------------------------------------------------------|
(33)       | AVERAGE NONGLOBAL VARIABLES PER SEGMENT               |
(33)       |    PARAMETER                                          |
(33)       |    LOCAL                                              |
           |=======================================================|
(36)       | PARAMETER PASSAGE TYPE PERCENTAGES :                  |
(36)       |    VALUE                                              |
(36)       |    REFERENCE                                          |
           |=======================================================|
(40)       | (SEGMENT,GLOBAL) ACTUAL USAGE PAIRS                   |
(34)       |    ENTRY                                              |
(35)       |       MODIFIED                                        |
(35)       |       UNMODIFIED                                      |
(34)       |    NONENTRY                                           |
(35)       |       MODIFIED                                        |
(35)       |       UNMODIFIED                                      |
(35)       |    MODIFIED                                           |
(35)       |    UNMODIFIED                                         |
           |-------------------------------------------------------|
(40)       | (SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS                 |
(34)       |    ENTRY                                              |
(35)       |       MODIFIED                                        |
(35)       |       UNMODIFIED                                      |
(34)       |    NONENTRY                                           |
(35)       |       MODIFIED                                        |
(35)       |       UNMODIFIED                                      |
(35)       |    MODIFIED                                           |
(35)       |    UNMODIFIED                                         |
           |-------------------------------------------------------|
(40)    *  | (SEGMENT,GLOBAL) USAGE PAIR REL.PERCENT.              |
(34)       |    ENTRY                                              |
(35)       |       MODIFIED                                        |
(35)       |       UNMODIFIED                                      |
(34)       |    NONENTRY                                           |
(35)       |       MODIFIED                                        |
(35)       |       UNMODIFIED                                      |
(35)       |    MODIFIED                                           |
(35)       |    UNMODIFIED                                         |
           *********************************************************
```

AVG. is an abbreviation for AVERAGE
REL.PERCENT. is an abbreviation for RELATIVE PERCENTAGE

elaborative product aspects

```
           *********************************************
(44)       | CYCLOMATIC COMPLEXITY :                     |
           |---------------------------------------------|
(45)       |    SIMPPRED-NCASE VARIATION :               |
(46)       |       TOTAL                                 |
(47)       |       #SEGS:CC>=10                          |
(46)       |       0.5 QUANTILE POINT VALUE              |
(48)       |       0.5 QUANTILE TAIL AVERAGE             |
(48)       |       0.7 QUANTILE POINT VALUE              |
(45)       |       0.7 QUANTILE TAIL AVERAGE             |
(43)       |       0.8 QUANTILE POINT VALUE              |
(43)       |       0.8 QUANTILE TAIL AVERAGE             |
(46)       |       0.9 QUANTILE POINT VALUE              |
(48)       |       0.9 QUANTILE TAIL AVERAGE             |
```

TABLE 1

```
(44)          |-----------------------------------------|
(45)          |   SIMPPRED-LOGCASE VARIATION :          |
(46)          |      TOTAL                              |
(47)          |      #SEGS:CC>=10                       |
(48)          |      0.5 QUANTILE POINT VALUE           |
(49)          |      0.5 QUANTILE TAIL AVERAGE          |
(50)          |      0.7 QUANTILE POINT VALUE           |
(51)          |      0.7 QUANTILE TAIL AVERAGE          |
(52)          |      0.8 QUANTILE POINT VALUE           |
(53)          |      0.8 QUANTILE TAIL AVERAGE          |
(54)          |      0.9 QUANTILE POINT VALUE           |
(55)          |      0.9 QUANTILE TAIL AVERAGE          |
              |-----------------------------------------|
(45)          |   COMPPRED-NCASE VARIATION :            |
(46)          |      TOTAL                              |
(47)          |      #SEGS:CC>=10                       |
(48)          |      0.5 QUANTILE POINT VALUE           |
(49)          |      0.5 QUANTILE TAIL AVERAGE          |
(50)          |      0.7 QUANTILE POINT VALUE           |
(51)          |      0.7 QUANTILE TAIL AVERAGE          |
(52)          |      0.8 QUANTILE POINT VALUE           |
(53)          |      0.8 QUANTILE TAIL AVERAGE          |
(54)          |      0.9 QUANTILE POINT VALUE           |
(55)          |      0.9 QUANTILE TAIL AVERAGE          |
              |-----------------------------------------|
(45)          |   COMPPRED-LOGCASE VARIATION :          |
(46)          |      TOTAL                              |
(47)          |      #SEGS:CC>=10                       |
(48)          |      0.5 QUANTILE POINT VALUE           |
(49)          |      0.5 QUANTILE TAIL AVERAGE          |
(50)          |      0.7 QUANTILE POINT VALUE           |
(51)          |      0.7 QUANTILE TAIL AVERAGE          |
(52)          |      0.8 QUANTILE POINT VALUE           |
(53)          |      0.8 QUANTILE TAIL AVERAGE          |
(54)          |      0.9 QUANTILE POINT VALUE           |
(55)          |      0.9 QUANTILE TAIL AVERAGE          |
              |=========================================|
(41)          | (SEGMENT,GLOBAL,SEGMENT) DATA BINDINGS :|
(42)      *   |   ACTUAL                                |
(43)          |      SUBFUNCTIONAL                      |
(43)          |      INDEPENDENT                        |
(42)      *   |   POSSIBLE                              |
(42)      *   |   RELATIVE PERCENTAGE                   |
              |=========================================|
(49)          | SOFTWARE SCIENCE QUANTITIES :           |
(50)          |   VOCABULARY                            |
(50)          |   LENGTH                                |
(50)          |   ESTIMATED LENGTH                      |
(50)          |   %DIFFERENCE(N,N^)                     |
(50)          |   VOLUME                                |
(50,99)       |   INTELLIGENCE CONTENT                  |
(50)          |   ESTIMATED BUGS                        |
              |-----------------------------------------|
(51)          |   1ST CALCULATION METHOD :              |
(50)          |      PROGRAM LEVEL                      |
(50)          |      DIFFICULTY                         |
(50,99)       |      POTENTIAL VOLUME                   |
(50)          |      LANGUAGE LEVEL                     |
(50)          |      EFFORT                             |
(50)          |      ESTIMATED TIME                     |
              |-----------------------------------------|
(51)          |   2ND CALCULATION METHOD :              |
(50)          |      PROGRAM LEVEL                      |
(50)          |      DIFFICULTY                         |
(50)          |      POTENTIAL VOLUME                   |
(50)          |      %DIFFERENCE(V*,I)                  |
(50)          |      LANGUAGE LEVEL                     |
(50)          |      EFFORT                             |
(50)          |      ESTIMATED TIME                     |
              *******************************************
```

(PROC)CALL aspect are each labeled and categorized from the viewpoint of implementation language construct frequencies. But the same metrics can also be considered from the viewpoint of segment invocation frequencies, warranting the inclusion of the two duplicate aspects INVOCATIONS\FUNCTIONS and INVOCATIONS\PROCEDURES as variants of the general INVOCATIONS aspect. Among the 197 programming aspects listed in Table 1, there are 8 pairs of duplicate aspects (identified in note 99 below), leaving 189 nonredundant aspects examined in the study. By definition, the data scores obtained for any pair of duplicate aspects will be identical, and thus the same statistical conclusions will be reached for both aspects. This redundancy must be kept in mind when evaluating the results of the experiments.

Brief explanatory notes about the programming aspects are given below, in the form of numbered paragraphs keyed to the list in Table 1, with definitions for the nontrivial or unfamiliar metrics. These notes usually supply loose explanations for the general concepts behind these programming aspects, before mentioning any restrictions or variations in how they were applied and measured in this study. Technical meanings for system- or language-dependent terms (e.g., module, segment, intrinsic, entry) also appear here. Since computer programming terminology is not standardized, the reader is cautioned against drawing inferences not based on this dissertation's definitions.

Explanatory Notes for the Programming Aspects

(1) A computer job step is a conceptually indivisible programmer-oriented activity that is performed on a computer at the operating system command level, is inherent to the software development effort, and involves a nontrivial expenditure of computer or human resources. Ideally

speaking, examples of job steps would include editing symbolic texts, compiling source modules, link-editing (or collecting) object modules, and executing entire programs; however, ocerations such as querying the operating system for status information or requesting access to on-line files would not qualify as job steps. In this study, consideration for the COMPUTER JOB STEPS aspect was limited exclusively to the activities of compiling source modules or executing entire programs, but not all of the activities so counted dealt with the final product (or logical predecessors thereof).

(2) A module compilation is an invocation of the implementation language processor on the source code of an individual module. In this study, only compilations of modules comprising the final software product (or logical predecessors thereof) are counted in the COMPUTER JOB STEPS\ MODULE COMPILATION aspect.

(3) All module compilations are (sub)categorized as either identical or unique depending on whether or not the source code compiled is textually identical to that of a previous compilation. During the development process, each unique compilation was necessary in some sense, while an identical compilation could conceivably have been avoided by saving the (relocatable) object module from a previous compilation for later reuse (except in the situation of undoing source code revisions after they have been tested and found to be erroneous or superfluous).

(4) A program execution is an invocation of a complete programmer-developed program (after the necessary compilation(s) and link-editing) upon some test data. In this study, only executions of programs composed of modules comprising the final product (or logical predecessors

thereof) are counted in the COMPUTER JOB STEPS\PROGRAM EXECUTION aspect.

(5) A miscellaneous job step is an auxiliary compilation or execution of something other than the final software product. In this study, the COMPUTER JOB STEPS\MISCELLANEOUS aspect counts exactly those activities included in the COMPUTER JOB STEPS aspect but not included in the COMPUTER JOB STEPS\MODULE COMPILATION or COMPUTER JOB STEPS\PROGRAM EXECUTION aspects.

(6) An essential job step is a computer job step that involves the final software product (or logical predecessors thereof) and could not have been avoided (by off-line computation or by on-line storage of previous compilations or results). In this study, the COMPUTER JOB STEPS\ESSENTIAL aspect is the sum of the COMPUTER JOB STEPS\MODULE COMPILATION\UNIQUE aspect plus the COMPUTER JOB STEPS\PROGRAM EXECUTION aspect.

(7) The AVERAGE UNIQUE COMPILATIONS PER MODULE aspect is a way of normalizing the COMPUTER JOB STEPS\MODULE COMPILATION\UNIQUE aspect.

(8) The MAXIMUM UNIQUE COMPILATIONS FOR ANY ONE MODULE aspect is another way of normalizing (by isolating the worst case) the COMPUTER JOB STEPS\MODULE COMPILATION\UNIQUE aspect.

(9) The program changes metric [Dunsmore & Gannon 77] is a measure of the total amount of textual revision made to program source code during the (postdesign) software development period. The rules for counting program changes are designed to identify individual conceptual changes algorithmically. Each occurrence of the following revisions

is counted as a single program change: modification of a single statement, insertion of contiguous statements, or modification of a single statement followed immediately by insertion of contiguous statements. However, the following revisions are not counted as program changes: deletion of contiguous statements, insertion of standard output statements or special compiler-provided debugging directives, and instances of lexical reformatting without syntactic/semantic alteration.

See Chapter V for further discussion of the program changes metric.

(10) A module is a separately compiled portion of the complete software system. In the implementation language SIMPL-T, a typical module is a collection of the declarations of several global variables and the definitions of several segments. In this study, only those modules which comprise the final product are counted in the MODULES aspect.

(11) A segment is a collection of source code statements, together with declarations for the formal parameters and local variables manipulated by those statements, that may be invoked as an operational unit. In the implementation language SIMPL-T, a segment is either a value-returning function (invoked via reference in an expression) or else a non-value-returning procedure (invoked via the CALL statement). The segment, function, and procedure of SIMPL-T correspond to the (sub)program, function, and subroutine of Fortran, respectively.

(12) The group of aspects named SEGMENT TYPE COUNTS gives the absolute number of programmer-defined segments of each type. The group of aspects named SEGMENT TYPE PERCENTAGES gives the relative percentage of each type of

segment, compared with the total number of programmer-defined segments. The latter group of aspects is a way of normalizing the former groups of aspects.

(13) Since in the implementation language SIMPL-T segment definitions occur within the context of a module, a natural way to normalize (or average) the raw counts of segments is provided. The AVERAGE SEGMENTS PER MODULE aspect represents the average size, in segments, of modules in the program.

(14) The LINES aspect counts every textual line of delivered source code in the final product, including comments, compiler directives, variable declarations, executable statements, etc.

(15) The STATEMENTS aspect counts only the executable constructs in the source code of the final product. These are high-level, structured-programming statements, including simple statements--such as assignment and procedure call--as well as compound statements--such as if-then-else and while-do--which have other statements nested within them. The implementation language SIMPL-T allows exactly seven different statement types (referred to by their distinguishing keyword or symbol) covering assignment (:=), alternation-selection (IF, CASE), iteration (WHILE, EXIT), and procedure invocation (CALL, RETURN). Input-output operations are accomplished via calls to intrinsic procedures.

(16) The group of aspects named STATEMENT TYPE COUNTS gives the absolute number of executable statements of each type. The group of aspects named STATEMENT TYPE PERCENTAGES gives the relative percentage of each type of statement, compared with the total number of executable statements.

The latter group of aspects is a way of normalizing the former groups of aspects.

(17) As mentioned above, the := symbol denotes the assignment statement. It assigns the value of the expression on the right hand side to the variable on the left hand side.

(18) Both if-then and if-then-else constructs are counted as IF statements. Each IF statement allows the execution of either the then- or else-part statements, depending upon its boolean expression.

(19) The CASE statement provides for selection from several alternatives, depending upon the value of an expression. In the implementation language SIMPL-T, exactly one of the alternatives (or an optional else-part) is selected per execution of a CASE, a list of constants is explicitly given for each alternative, and selection is based upon the equality of the expression value with one of the constants. (These constants are referred to as "case labels"; these alternatives, as "case branches.") A case construct with n alternatives is logically and semantically equivalent to a series of n nested if-then-else constructs.

(20) The WHILE statement is the only iteration or looping construct provided by the implementation language SIMPL-T. It allows the statements in the loop body to be executed repeatedly (zero or more times) depending upon a Boolean expression which is reevaluated at every iteration; the loop may also be terminated via an EXIT statement. Each WHILE statement may be optionally labeled with a designator (referenced by EXIT statements) which uniquely identifies it from other nested WHILE statements.

(21) The EXIT statement allows the abnormal termination of iteration loops by unconditional transfer of control to the statement immediately following the WHILE statement. Thus it is a very restricted form of GOTO. This exiting may take place from any depth of nested loops, since the EXIT statement may optionally name a designator which identifies the loop to be exited; without such a designator only the immediately enclosing loop is exited.

(22) Since there are two types of segments in the implementation language SIMPL-T, there are two types of "calls" or segment invocations. Procedures are invoked via the CALL statement, and functions are invoked via reference in an expression. The counts for these separate constructs are reported separately as the (PROC)CALL and FUNCTION CALL aspects, and jointly as the INVOCATIONS aspect.

(23) Intrinsic means provided and defined by the implementation language; nonintrinsic means provided and defined by the programmer. These terms are used to distinguish built-in procedures or functions (which are supported by the compiler and utilized as primitives) from segments (which are written by the programmer). Nearly all of the intrinsic procedures in the implementation language SIMPL-T perform input-output operations and external data file manipulations. All of the intrinsic functions in SIMPL-T perform data type coercions and character string operations.

(24) The RETURN statement allows the abnormal termination of the current segment by unconditional resumption of the previously executing segment. Thus it is another very restricted form of GOTO. Within a function, a RETURN statement must specify an expression, the value of which becomes the value returned for the function

invocation. Within a procedure, a RETURN statement must not specify such an expression. Additionally, a simple RETURN statement is optional at the textual end of procedures; it will be implicitly assumed if not explicitly coded. In this study, the total number of explicitly coded and implicitly assumed RETURN statements, both from functions and procedures combined, is counted.

(25) The AVERAGE STATEMENTS PER SEGMENT aspect provides a way of normalizing the number of statements relative to their natural enclosure in a program, the segment. The measure also represents the average length, in executable statements, of segments in the program.

(26) In the implementation language SIMPL-T, both simple (e.g., assignment) and compound (e.g., if-then-else) statements may be nested inside other compound statements. A particular nesting level is associated with each statement, starting at 1 for a statement at the outermost level of each segment and increasing by 1 for successively nested statements. Nesting level can be displayed visually via proper and consistent indentation of the souce code listing.

(27) The DECISIONS aspect is the sum of the numbers of IF, CASE, and WHILE statements within the program's source code. Each of these statements represents a unique (possibly repeated) run-time decision coded by the programmer. Because the implementation language SIMPL-T has only structured control structures, this aspect is closely related to the cyclomatic complexity metrics discussed below.

(28) Tokens are the basic syntactic entities--such as keywords, operators, parentheses, identifiers, etc.--that

occur in a program statement. The average number of tokens per statement may be viewed as an indication of how much "information" a typical statement contains, how "powerful" a typical statement is, or how concisely the statements are coded.

(29) An *invocation* is the syntactic occurrence of a construct by which either a programmer-defined segment or a built-in routine is invoked from within another segment; both procedure calls and function references are counted as INVOCATIONS. They are (sub)categorized by the type (i.e., function or procedure, nonintrinsic or intrinsic) of segment or routine being invoked.

(30) The group of aspects named AVERAGE INVOCATIONS PER (CALLING) SEGMENT represents one way to normalize the absolute number of invocations. These aspects reflect the average number of calls to programmer-defined segments and built-in routines from a programmer-defined segment. They are (sub)categorized by the type of segment or routine being invoked.

(31) The group of aspects named AVERAGE INVOCATIONS PER (CALLED) SEGMENT represents another way to normalize the absolute number of invocations. These aspects reflect the average number of calls to a programmer-defined segment from other segments. They are (sub)categorized by the type (i.e., function or procedure) of segment being invoked.

(32) A *data variable* is an individually named scalar or structure. The implementation language SIMPL-T provides:
(a) three data *types* for scalars--integer, character, and (varying-length) string;
(b) one kind of data *structure* (besides scalar)--single dimensional array, with zero-origin subscript range;

and

(c) several levels of scope (as explained in note 33 below) for data variables.

In addition, all data variables in a SIMPL-T program must be explicitly declared, with attributes fully specified. The DATA VARIABLES aspect counts each data variable declared in the final software product once, regardless of its type, structure, or scope. Note that each array is counted as a single data variable.

(33) In the implementation language SIMPL-T, data variables can have any one of essentially four levels of scope--entry global, nonentry global, parameter, and local--depending on where and how they are declared in the program. Note that the notion of scope deals only with static accessibility by name; the effective accessibility of any variable can always be extended by passing it as a parameter between segments. The scope levels are explained here (and presented in the aspect (sub)categorizations) via a hierarchy of distinctions.

The primary distinction is between global and nonglobal. Global variables are accessible by name to each of the segments in the module in which they are declared. Nonglobal variables are accessible by name only to the single segment in which they are declared.

Global variables are secondarily distinguished into entry and nonentry categories. Entry globals may be accessible by name to each of the segments in several modules (as explained in note 34 below). Nonentry globals are accessible by name only within the module in which they are declared.

Nonglobal variables are secondarily distinguished into formal parameters and locals. Formal parameters are accessible by name only within the enclosing (called) segment, but their values are related to the calling segment

(as explained in note 36 below). Locals are accessible by name only within the enclosing segment, and their values are completely isolated from any other segment.

(34) Entry means that the data variable (explicitly declared as ENTRY in one module) is accessible from within other separately compiled modules (in which it must be explicitly declared as EXTernal). Nonentry means that the data variable is accessible only within the module in which it is declared. In this study, these terms are used only in reference to global variables, although the implementation language SIMPL-T handles the accessibility of segments across modules in the same way.

Although the implementation language SIMPL-T does allow the EXTernal attribute to be declared locally so that just the enclosing segment has access to an identifier declared as ENTRY in another module, this feature is seldom used; it never occurred in any of the final software products examined in this study.

(35) Modified means referred to, at least once in the program source code, in such a manner that the value of the data variable might be (re)set when (and if) the appropriate statements were to be executed. Data variables can be (re)set only by (a) being the "target" of an assignment statement, (b) being passed by reference to a programmer-defined segment or built-in routine, or (c) being named in an "input statement." (This third case is really covered by the second case since all the "input statements" in SIMPL-T are actually calls to certain intrinsic procedures with passed-by-reference parameters.) Unmodified means referred to, throughout the program source code, in such a manner that the value of the data variable could never be (re)set during execution. These terms refer only to global data variables.

Any global variable is allowed to have an initial value (constants only) specified in its declaration. Globals which are initialized but unmodified are especially useful in SIMPL-T programs, serving as "named constants."

(36) The implementation language SIMPL-T allows two types of parameter passage. Pass-by-value means that the value of the actual argument is copied (upon invocation) into the corresponding formal parameter (which thereafter behaves like a local variable for all intents and purposes); the effect is that the called routine cannot modify the value of the calling segment's actual argument. Pass-by-reference means that the address of the actual argument (which must be a variable rather than an expression) is passed (upon invocation) to the called routine; the effect is that any changes made by the called routine to the corresponding formal parameter will be reflected in the value of the calling segment's actual argument (upon return). In SIMPL-T, formal parameters that are scalars are normally (default) passed by value, but they may be explicitly declared to be passed by reference; formal parameters that are arrays are always passed by reference.

(37) The group of aspects named DATA VARIABLE SCOPE COUNTS gives the absolute number of declared data variables according to each level of scope. The group of aspects named DATA VARIABLE SCOPE PERCENTAGES gives the relative percentage of variables at each scope level, compared with the total number of declared variables. The latter group of aspects is a way of normalizing the former groups of aspects.

(38) A natural way to normalize (or average) the raw counts of data variables is provided, since data variable declarations in the implementation language SIMPL-T may only

appear in certain contexts within the program: globals in the context of a module and nonglobals in the context of a segment. The group of aspects named AVERAGE GLOBAL VARIABLES PER MODULE represents the average number of globals declared for a module. The group of aspects named AVERAGE NONGLOBAL VARIABLES PER SEGMENT represents the average number of nonglobals declared for a segment.

(39) Since there are two types of parameter passing mechanisms in the implementation language SIMPL-T (as explained in note 36 above), it is desirable to normalize their raw frequencies into relative percentages, indicating the programmer's degree of "preference" for one type or the other. The group of aspects named PARAMETER PASSAGE TYPE PERCENTAGES gives the percentages of each type of parameter relative to the total number of parameters declared in the program.

(40) A segment-global usage pair (p,r) is an instance of a global variable r being used by a segment p (i.e., the global is either modified (set) or accessed (fetched) at least once within the statements of the segment). Each usage pair represents a unique "use connection" between a global and a segment. In this study, segment-global usage pairs were (sub)categorized by the type (i.e., entry or nonentry, modified or unmodified) of global data variable involved and were counted in three different ways.

First, the (SEGMENT,GLOBAL) ACTUAL USAGE PAIRS aspects count the absolute numbers of realized usage pairs (p,r) : the global variable r is actually used by segment p . They represent the frequencies of use connections realized within the program. Second, the (SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS aspects count the absolute numbers of potential usage pairs (p,r) , given the program's global variables

and their declared scope: the scope of global variable r merely contains segment p , so that p could potentially modify or access r . These counts of possible usage pairs are computed as the sum of the number of segments in each global's scope. They represent a sort of "worst case" frequencies of use connections. Third, the (SEGMENT,GLOBAL) USAGE PAIR RELATIVE PERCENTAGE aspects are a way of normalizing the number of usage pairs since these measures are ratios (expressed as percentages) of actual usage pairs to possible usage pairs. They represent the frequencies of realized use connections relative to potential use connections. These usage pair relative percentage metrics are empirical estimates of the likelihood that an arbitrary segment uses (i.e., sets or fetches the value of) an arbitrary global variable.

In some sense, all three types of aspects dealing with segment-global usage pairs (actual, possible, and relative percentage) reflect quantifiable characteristics of "data modularization" within a program, i.e., the static organization of data definitions and references within segments and modules. In particular, the possible usage pairs aspects reflect the general degree of encapsulation enforced by the implementation language for global variables. Moreover, the usage pair relative percentage aspects reflect the general degree of "globality" for global variables, i.e., the extent to which globals are actually used by those segments that could possibly do so.

(41) A segment-global-segment <u>data binding</u> [Stevens, Myers & Constantine 74, pp. 118-119] (p,r,q) is defined as an occurrence of the following arrangement in a program: a segment p modifies (sets) a global variable r that is also accessed (fetched) by a segment q , with p different from q . The (SEGMENT,GLOBAL,SEGMENT) DATA BINDINGS aspects count these unique communication paths between pairs

of segments via global variables. These aspects thus reflect the degree of one form of connectivity within a program.

See Chapter V for further discussion of the data bindings metrics.

(42) In this study, segment-global-segment data bindings were counted in three different ways: ACTUAL, POSSIBLE, and RELATIVE PERCENTAGE. First, the DATA BINDINGS\ACTUAL aspect counts the total number of data bindings actually coded in the program, reflecting the degree of realized connectivity. Second, the DATA BINDINGS\POSSIBLE aspect counts the total number of data bindings that could possibly be allowed, given the program's organizational structure. It reflects the degree of potential connectivity. Third, the DATA BINDINGS\RELATIVE PERCENTAGE aspect is the ratio (expressed as a percentage) of actual data bindings to possible data bindings, reflecting the normalized degree of realized connectivity relative to potential connectivity.

See Chapter V for further discussion of the data bindings metrics.

(43) Actual data bindings are (sub)categorized depending on the invocation relationship between the two segments. A data binding (p,r,q) is subfunctional if either of the two segments p or q can invoke the other, whether directly or indirectly, as a "subroutine." A data binding (p,r,q) is independent if neither of the two segments p or q can invoke the other, whether directly or indirectly.

See Chapter V for further discussion of the data bindings metrics.

(44) Cyclomatic complexity [McCabe 76] is a graph-

theoretic measure of control-flow complexity. For an implementation language with only structured control structures (such as SIMPL-T), this measure is dependent only on the number of predicates (i.e., Boolean expressions governing flow of control) in the source code. The cyclomatic complexity $v(P)$ of a program P with Π predicates strewn among s segments is computed as

$$v(P) = \Pi + s ;$$

the cyclomatic complexity $v(S)$ of a segment S with π predicates is computed as

$$v(S) = \pi + 1 .$$

See Chapter V for further discussion of the cyclomatic complexity metrics.

(45) Four definitional variations of the basic cyclomatic complexity measure were examined in this study in order to explore alternatives for identifying predicates and for handling case statement constructs. Under the SIMPPRED alternative, simple Boolean subexpressions joined by and or or connectives are each counted as predicates. Under the COMPPRED alternative, only each complete Boolean expression is counted as a predicate. Under the NCASE alternative, each case statement construct is counted as contributing n predicates, where n is the number of "cases" involved. Under the LOGCASE alternative, each case statement construct is counted as contributing $\lfloor \log_2(n) \rfloor$* predicates, thus giving a discount for case statement constructs relative to series of nested ifthenelse constructs.

See Chapter V for further discussion of the cyclomatic complexity metrics.

(46) For each of the definitional variations, the CYCLOMATIC COMPLEXITY\...\TOTAL aspect measures the

---

* The notation $\lfloor x \rfloor$ signifies the greatest integer less than or equal to x.

cyclomatic complexity of the entire program. It is simply the sum of cyclomatic complexity values for the individual segments comprising the program.

See Chapter V for further discussion of the cyclomatic complexity metrics.

(47) For each of the definitional variations, the CYCLOMATIC COMPLEXITY\...\#SEGS:CC>=10 aspect counts the number of segments in the program whose cyclomatic complexity values equal or exceed the threshold value 10.

See Chapter V for further discussion of the cyclomatic complexity metrics.

(48) For each of the definitional variations, a common descriptive statistic of the empirical distribution of cyclomatic complexity values from the individual segments comprising an entire an entire program was used as a vehicle for measuring the general level of cyclomatic complexity within the relatively nontrivial segments of the program. This descriptive statistic, known as a quantile [Conover 71, pp. 31-32, pp. 72-73], can be loosely described (in the discrete case) as the value (of the random variable in question) corresponding to a particular fixed probability level on the cumulative relative frequency curve (representing the distribution of that random variable). The CYCLOMATIC COMPLEXITY\...\ f QUANTILE POINT VALUE aspects are defined to measure the largest integer x such that the fraction of cyclomatic complexity values which are less than x is less than or equal to the fixed fraction f. The CYCLOMATIC COMPLEXITY\...\ f QUANTILE TAIL AVERAGE aspects are defined to measure the average of cyclomatic complexity values greater than or equal to the f quantile point value. Several particular quantiles were examined in this study: the 0.5 quantile is closely related to the distribution's median, and the 0.7, 0.8, and 0.9 quantiles

provide a series of increasingly smaller tails of the distribution.

See Chapter V for further discussion of the cyclomatic complexity metrics.

(49) According to software science theory [Halstead 77], several interesting quantities can be computed from the source code of a program and used to measure characteristics of both the abstract algorithm and its expression as implemented. All of these software science quantities are computed in terms of the number of conceptually unique "operators" and "operands" and the total occurrences of such "operators" and "operands" within a program. In this study, these "operators" and "operands" were identified syntactically according to a set of rules established for the implementation language SIMPL-T.

See Chapter V for further discussion of the software science metrics.

(50) Given the basic parameters of software science:

total "operator" count N_1
total "operand" count N_2
unique "operator" count η_1
unique "operand" count η_2
unique potential "operand" count η_2^*

the following formulas define the software science quantities examined in this study:

VOCABULARY	$\eta = \eta_1 + \eta_2$
LENGTH	$N = N_1 + N_2$
ESTIMATED LENGTH	$\hat{N} = (\eta_1 * \log_2(\eta_1)) + (\eta_2 * \log_2(\eta_2))$
%DIFFERENCE(N,$\hat{N}$)	$= (\lvert\hat{N} - N\rvert) / (N)$
VOLUME	$V = N * \log_2(\eta)$
POTENTIAL VOLUME	$V^* = (2 + \eta_2^*) * \log_2(2 + \eta_2^*)$
PROGRAM LEVEL	$L = V^* / V$
DIFFICULTY	$D = 1 / L$

INTELLIGENCE CONTENT $I = (2 / \eta_1) * (\eta_2 / N_2) * V$

"DIFFERENCE(V*,I) $= (|I - V^*|) / (V^*)$

LANGUAGE LEVEL $\lambda = L * V^*$

EFFORT $E = V * D$

ESTIMATED TIME $T = E / S$

ESTIMATED BUGS $B = L * E / E_o$

where S and E_o are psychologically determined constants.

See Chapter V for further discussion of the software science metrics.

(51) Two different calculation methods were employed in the study to compute the subset of software science quantities whose exact values cannot be obtained directly (via the defining formulas) from a program's source code. These calculation methods each rely upon a different estimation technique to obtain approximate values for these quantities. The 1ST CALCULATION METHOD relies upon the commonly accepted theoretical estimate of the program level quantity; the 2ND CALCULATION METHOD relies upon an internally applied empirical estimate of the language level quantity.

See Chapter V for further discussion of the software science metrics.

(99) Several instances of duplicate programming aspects exist in the Table 1 listing. That is, some logically unique aspects reappear with another label, for reasons explained above. Listed below are the pairs of duplicate programming aspects that were considered in this study:

1.	FUNCTION CALLS	<=>	INVOCATIONS\FUNCTION
2.	NONINTRINSIC	<=>	NONINTRINSIC
3.	INTRINSIC	<=>	INTRINSIC
4.	STATEMENT TYPE COUNTS\ (PROC)CALL	<=>	INVOCATIONS\PROCEDURE
5.	NONINTRINSIC	<=>	NONINTRINSIC

6.	INTRINSIC	<=>	INTRINSIC
7.	AVERAGE INVOCATIONS PER (CALLING) SEGMENT\ NONINTRINSIC	<=>	AVERAGE INVOCATIONS PER (CALLED) SEGMENT
8.	SOFTWARE SCIENCE QUANTITIES\INTELLIGENCE CONTENT	<=>	SOFTWARE SCIENCE QUANTITIES\1ST CALCULATION METHOD\POTENTIAL VOLUME

By definition, the data scores obtained for any pair of duplicate aspects will be identical, and thus the same statistical conclusions will be reached for both aspects.

CHAPTER V

V. DISCUSSION OF ELABORATIVE METRICS

This chapter provides an in-depth discussion of the elaborative programming aspects examined in the study. The material is presented, in a tutorial fashion, in order to motivate their appeal as software metrics and to explain how they might be interpreted. The reader who is acquainted with one or more of these measures might consider skimming the corresponding sections.

Program Changes

The program changes metric pertains to textual revisions made to program source code during development, from the time a program is first presented to the computer system, to the completion of the project. The metric's definition is framed so that one program change approximates one conceptual change to the program. The following rules for identifying program changes are reproduced from [Dunsmore 78, pp. 19-20]:

> "The following text changes to a program represent one program change:
>
> 1. One or more changes to a single statement.
>   (Even multiple character changes to a statement represent mental activity with only a single abstract instruction.)
>
> 2. One or more statements inserted between existing statements.
>   (The contiguous group of statements inserted probably corresponds to the concrete statements that represent a single abstract instruction.)
>
> 3. A change to a single statement followed by the insertion of new statements.
>   (This instance probably represents a discovery that an existing statement is insufficient and that it must be altered and supplemented by additional statements to implement the abstract instruction involved.)
>
> "However, the following text changes to a program are not counted as program changes:

1. The deletion of one or more statements.
   (Deleted statements must usually be replaced by other statements elsewhere. The inserted statements are counted; counting deletions as well would give double weight to such a change. Occasionally statements are deleted but not replaced; these were probably being used for debugging purposes and their deletion requires litte mental activity.)

2. The insertion of standard output statements or insertion of special compiler-provided debugging statements.
   (These are occasionally inserted in a wholesale fashion during debugging. When the problem is found, these are then all removed, and the necessary program change takes place.)

3. The insertion of blank lines, insertion of comments, revision of comments, and reformatting without alteration of existing statements.
   (These are all judged to be cosmetic in nature.)"

Program changes are counted algorithmically by comparing the source code from each pair of consecutive compilations of a module (or logical predecessor thereof) and applying the identification rules. Thus the total number of program changes is a measure of the amount of textual revision to source code during (postdesign) system development.

The program changes metric may be interpreted as a programming complexity measure, because textual revisions are usually necessitated by errors encountered while building, testing, and debugging software. Independent research [Dunsmore & Gannon 77] has demonstrated a high (rank order) correlation between total program changes (as counted automatically according to a specific algorithm) and total error occurrences (as tabulated manually from exhaustive scrutiny of source code and test results) during software implementation in the SIMPL-T programming language. Thus empirical evidence justifies consideration of the program changes metric as a direct measure of the relative number of programming errors encountered outside of design work. It is reasonable to assume that each textual revision entails some expenditure of the programmer's effort (e.g., planning the revision, editing source code on-line). In

that sense, this metric may also be considered an indirect measure of the level of human effort devoted to implementation.

Cyclomatic Complexity

Control-flow complexity may be measured in terms of cyclomatic complexity [McCabe 76], a graph-theoretic metric that is independent of physical size (i.e., insertion or deletion of function statements leaves the measure unchanged) and dependent only on the decision structure of a program. The cyclomatic number $v(G)$ of a graph G having n nodes, e edges, and p connected components is defined as

$$v(G) = e - n + p .$$

In a strongly connected graph, the cyclomatic number is equal to the minimum number of basis paths from which all other paths may be constructed as linear combinations in an edge-algebraic fashion (see [McCabe 76] for details). By modeling the control flow of a program as a graph in the traditional manner, the cyclomatic complexity measure is defined to be the cyclomatic number of the graph corresponding to the program's flow of control.

For a structured language like SIMPL-T, it is not necessary to construct a control-flow graph in order to measure a program's cyclomatic complexity. The measure can be computed directly from the source code simply by counting the number of predicates (i.e., Boolean expressions governing control flow), since the predicates of the program correspond exactly to the binary-branching decision points of the control-flow graph. It is easily shown, using a lemma proven in [Mills 72], that for a segment S with π predicates the segment's cyclomatic complexity is

$v(S) = \pi + 1$

and for a program P with Π predicates strewn among s segments the program's cyclomatic complexity is

$v(P) = \Pi + s$.

This measure originated as an absolute count of the maximum number of linearly independent execution paths through a segment, in the graph-theoretic edge-algebraic sense alluded to above. Since each of these paths merits individual testing, the measure was proposed to serve as a quantitative indicator of the difficulty of testing a given segment to a certain degree of thoroughness. Testability is clearly an issue closely related to software complexity in general, and a program's cyclomatic complexity may be viewed as one quantitative measure of its control-structure complexity.

Definitional Variations

Several variations of the basic cyclomatic complexity measure were considered, because there are at least two definitional issues for which intuitively motivated alternatives lead to meaningful variations.

One of these issues is the weighting given to instances of case statement constructs. The original definition of cyclomatic complexity views a case statement as the semantically equivalent series of nested ifthenelse statements: each case statement contributes n units of cyclomatic complexity, where n is the number of individual "cases" involved. It can be argued, however, that a case statement deserves a smaller contribution to cyclomatic complexity since its inherent uniformity and readability have a moderating effect on programmer-perceived complexity (relative to an explicit series of nested ifthenelse

statements). One reasonable alternative views each case statement as contributing $\lfloor \log_2(n) \rfloor$* units of cyclomatic complexity, where n is the number of individual "cases" involved. This logarithmic weighting is appropriate since a case statement's moderating effect seems to increase with the number of "cases" involved.

The other issue is the manner of counting predicates. The original definition counts simple (sub)predicates individually, so that the compound predicate

(I < J) and ((A(I) = A(J)) or (not SORTED))

would contribute three units of cyclomatic complexity, for example. An alternative definition considers each complete predicate as an indivisible part of a program, contributing one unit of cyclomatic complexity. The motivation is that the complete predicate represents a single abstract condition governing the flow of control. Note that this issue is the basis for a proposed extension [Myers 77] to the original cyclomatic complexity measure. This issue also affects the way individual "cases" of a case statement construct are identified and counted. The original definition counts each case label separately, since multiple case labels on the same case branch are semantically equivalent to simple predicates joined by or's to form the Boolean expression governing the case branch. The alternative defintion counts only the case branches themselves, regardless of case label multiplicity. In parallel with the motivation given above, multiple case labels on a case branch represent a single abstract condition governing that branch (e.g., the set of case labels 0, 1, 2, ..., 9 may be abstracted to digit).

This study examined the four variations of cyclomatic

---

* The notation $\lfloor x \rfloor$ signifies the greatest integer less than or equal to x.

complexity defined as follows for the SIMPL-T programming language:

SIMPPRED-NCASE -- Simple predicates contribute 1 unit; case statements contribute 1 unit for each case label.

SIMPPRED-LOGCASE -- Simple predicates contribute 1 unit; case statements contribute $\lfloor \log_2(n) \rfloor$ units, where n is the number of case labels.

COMPPRED-NCASE -- Compound predicates contribute 1 unit; case statements contribute 1 unit for each case branch; multiple case labels on the same case branch are disregarded.

COMPPRED-LOGCASE -- Compound predicates contribute 1 unit; case statements contribute $\lfloor \log_2(n) \rfloor$ units, where n is the number of case branches; multiple case labels on the same case branch are disregarded.

Note that the SIMPPRED-NCASE variation of cyclomatic complexity is McCabe's original measure.

Techniques for Application

There are several ways to apply the cyclomatic complexity measure (or variations thereof) to an entire program in order to obtain a metric for its overall control-flow complexity. First of all, the metric is defined directly for a program composed of individual segments: a program's total cyclomatic complexity is simply the sum of its segments' cyclomatic complexities. However, this total cyclomatic complexity measure is not particularly useful as a basis for comparing entire programs because it is, in a certain sense, insensitive to the program's modularization. As a metric, the total cyclomatic complexity of a program is (by definition) a linear function in two variables, the number of predicates and the number of segments. A subtle

trade-off relationship exists between these two variables, such that substantial fluctuation in the metric's value can arise from simpleminded changes to a program's modularization alone.

A better comparison of entire programs is afforded by focusing attention upon the cyclomatic complexity values of individual segments and upon instances of segments with high values of the metric. McCabe originally proposed the number 10 as a reasonable threshold value for a segment's cyclomatic complexity. Segments exceeding this threshold need to be recoded or decomposed into smaller segments in order to attain an acceptable level of testability and control-flow complexity. Hence, a second way to apply the cyclomatic complexity measure to an entire program is to count the number of segments whose cyclomatic complexity value exceeds this threshold. In this case, the basis for comparing entire programs is the frequency of segments with unacceptably high cyclomatic complexity.

Finally, it would be desirable to compare the full spectrum of cyclomatic complexity values for the individual segments of one program against that of another program, since considerable diversity often exists. Programs typically contain several small segments with very low cyclomatic complexity values (e.g., a function to compute the average of a vector) and a few large segments with high cyclomatic complexity values. Being easily understood and tested, the small segments are relatively inconsequential, while the large segments contain the substance of the program and contribute most of the consequential control-flow complexity. Ideally, one wishes to disregard the "smaller" cyclomatic complexity values and summarize the magnitude and frequency of the "larger" cyclomatic complexity values via a single quantitative indicator, but

do so in a flexible, normalized fashion, where "smaller" and "larger" are determined relative to one another within each program.

This ideal can be approximated by means of the quantiles of the empirical distribution of cyclomatic complexity values across the segments comprising the program (see Figure 1). Quantiles are a standard tool from descriptive statistics [Conover 71, pp. 31-32, pp. 72-73], commonly used to summarize the "shape" and "position" of a distribution function, especially its upper tail region. Both the quantile point value (i.e., the largest integer x such that the fraction of cyclomatic complexity values which are less than x is less than or equal to some fixed fraction) and the quantile tail average (i.e., the average of cyclomatic complexity values greater than or equal to the quantile point value) are normalized ways to quantify just how high the cyclomatic complexity is for the relatively nontrivial segments of a program. Several different quantiles were examined: the 0.5 quantile is closely related to the median of the distribution, and the 0.7, 0.8, and 0.9 quantiles provide a series of increasingly smaller tails of the distribution. Thus, the basis for this third comparison of entire programs is a series of quantitative descriptors of the empirical distribution of cyclomatic complexity values within a program.

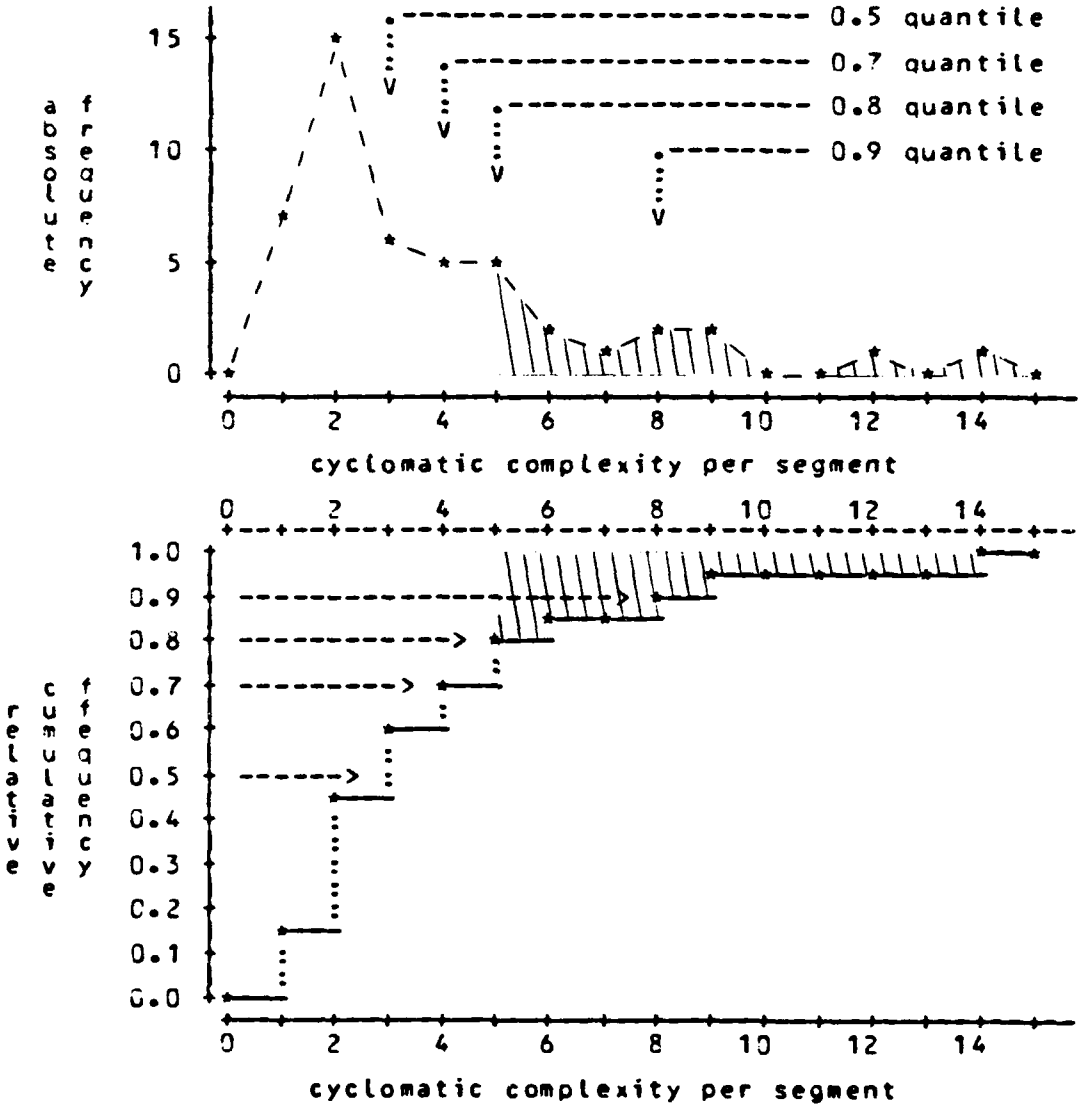
Data bindings

The data bindings metrics [Stevens, Myers & Constantine 74; Basili & Turner 75; Turner 76] originated as a way to quantify a certain kind of connectivity (i.e., directed communication between segments via global variables) within a program. Their motivation is based on the intuitive

FIGURE 1

Figure 1. Frequency Distribution of Cyclomatic Complexity

Both the absolute and the relative-cumulative frequency distribution of cyclomatic complexity values from 47 segments comprising an entire program are plotted. The tail region associated with the 0.8 quantile is shaded on each plot.



principle that the logical complexity of a composite system is a function of the multiplicity of connections among its component parts (cf. [Simon 69]).

A segment-global-segment *data binding* (p,r,q) is an occurrence of the following arrangement in a program: a segment p modifies a global variable r that is also accessed by a segment q, with segment p different from segment q. The existence of a data binding (p,r,q) suggests that the behavior of segment q is probably dependent on the performance of segment p through the data variable r, whose value is set by p and fetched by q. The binding (p,r,q) is different from the binding (q,r,p) which may also exist; occurrences such as (p,r,p) are not counted as data bindings. Thus each data binding represents a unique communication path between a pair of segments via a global variable. As a *metric*, the total number of segment-global-segment data bindings reflects the degree of that kind of connectivity within a program.

Data bindings may be counted in three different ways: actual, possible, and relative percentage. (Bear in mind that, since these measures are determined statically from the source code, the terms 'actual' and 'possible' refer to a program's syntactic form only.)

First, the *actual* count is the absolute number of data bindings (p,r,q) actually coded in the program: segment p contains a statement modifying global variable r, and segment q contains a statement accessing r. This count of actual data bindings represents the degree of realized connectivity in the program. Second, the *possible* count is the absolute number of data bindings (p,r,q) that could possibly be allowed under the program's structure of segment

definitions and global variable declarations: the scope of global variable r merely contains both segment p and segment q, so that segment p could potentially modify r and segment q could potentially access r. This count of possible data bindings represents the degree of potential connectivity, in a "worst-case" sense. It is computed as the sum of terms s*(s-1) for each global variable, where s is the number of segments in that global's scope; thus, it is heavily influenced (numerically speaking) by the sheer number of segments in a program. Third, the relative percentage is a way of normalizing the absolute numbers of data bindings, since it is simply the quotient (expressed as a percentage) of actual data bindings divided by possible data bindings. It represents the degree of realized connectivity relative to potential connectivity.

Actual data bindings may also be subcharacterized on the basis of the invocation relationship between the two segments. A data binding (p,r,q) is subfunctional if either of the two segments p or q can invoke the other, whether directly or indirectly (via a chain of intermediate invocations involving other segments). In this situation, the functioning of the one segment may be viewed as contributing to the overall functioning of the other segment. A data binding (p,r,q) is independent if neither of the two segments p or q can invoke the other, whether directly or indirectly. The transitive closure of the call graph among the segments of a program is employed to make this distinction between subfunctional and independent data bindings.

In some sense, all three measures dealing with segment-global-segment data bindings--actual, possible, and relative percentage--reflect quantifiable characteristics of a program's "data modularization" (i.e., the static

organization of data definitions and references within segments and modules).

In particular, the possible data bindings metric reflects the general degree of encapsulation enforced by the implementation language for global variables. One can imagine two extremes of encapsulation for the same collection of global variables and segments. On the one hand, the program could be written (in the implementation language SIMPL-T) as a single module containing all the segments, with each global potentially accessible from every segment. This modularization would maximize (explosively so, due to the squaring of the number of segments) the number of possible data bindings. On the other hand, the program could be written (in the implementation language SIMPL-T) as several modules, one for each segment, with appropriate ENTRY and EXTernal declarations to provide each segment with potential access to exactly those globals it actually uses. This modularization would minimize the number of possible data bindings (to precisely the number of actual data bindings).

Moreover, the data bindings relative percentage metric also reflects the general degree of (operational) "globality" for the global variables declared in a program, i.e., the extent to which globals are actually modified (set) and accessed (fetched) by those pairs of segments that could possibly do so. One can imagine two different situations in which the relative percentage of data bindings for a small set of otherwise equivalent global variables (say, an array and an integer) would be extremely high and extremely low, respectively. On the one hand, this global array and global integer could be serving as a stack, and nearly every segment that refers to these globals could be both popping the stack to examine its contents and pushing

new items onto it. Here the two global variables are quite central to the overall operation of that collection of segments; their data binding relative percentage would be close to one. On the other hand, this global array and global integer could be serving as a buffer for a varying-length vector that is initially produced (set) by one segment and nondestructively consumed (fetched only) by several other segments. Here the two global variables are rather incidental to the overall operation of that collection of segments, serving merely as a convenient medium for disseminating information (which could also have achieved via parameter passing); their data binding relative percentage would be close to zero.

Software Science Quantities

The software science quantities are a set of metrics based upon the tenets of software science [Halstead 77], as pioneered by Halstead and his colleagues. Billed as

> "... a branch of experimental and theoretical science dealing with the human preparation of computer programs and other types of written material ...,"*

software science is concerned with measurable attributes of algorithms or programs and with mathematical relationships among those attributes. Software science is characteristically actuarial in nature: its measures and relationships may be inaccurate when applied to individual programs, but they become surprisingly more accurate when applied to large numbers of programs, such as are found in large software development projects.

The software science quantities are all defined in

---

* The blocked quotations throughout this section are taken from [Halstead 77].

terms of certain frequencies of so-called "operators" and "operands" appearing within an algorithm's functional specification or a program's source code implementation. Some of the quantities (e.g., vocabulary, length, volume) are purely descriptive and provide the building-blocks of the theory. A few (e.g., estimated length) are predictive of other descriptive quantities within the theory. Several (e.g., program level, language level, effort) claim to be quantifications of fundamentally qualitative and intuitive concepts. Still other quantities (e.g., estimated time, estimated bugs) purport to measure--under ideal conditions--externally observable and quantifiable programming phenomena.

Identification Criteria

The criteria for identifying "operators" and "operands" (and their uniqueness) are important since they are the foundation for measuring the software science quantities. However, this identification is an area not clearly addressed by the theory. Except for the Fortran programming language, in which most of the pioneering work was done, explicit standards or guidelines for identification do not exist. For another language, a researcher can only attempt to adapt and extend the principles that he personally judges to be behind the Fortran work. The following "operator/operand" identification criteria were designed for the SIMPL-T programming language:

1. In general, only the portion of source code pertaining to executable statements (after expansion of all DEFINE-macros) is considered.
2. Constants and data variable identifiers are natural "operands." Data structures (e.g., arrays, files) are considered single objects and not decomposed into components.

3. The input stream file and the output stream file are counted as "operands," with implicit occurrences recognized for each operation on these files.
4. The normal prefix unary and infix binary operator symbols (i.e., for arithmetic, logical, and character-string operations) are natural "operators."
5. The intrinsic procedures (e.g., READ, WRITE, REWIND), type-coercion functions, and input-output operation keywords (e.g., EJECT, SKIP) are "operators."
6. Segment invocations (i.e., procedure calls and function references) are "operators."
7. Different types of statements or constructs are considered individual "operators," as follows:

   := (assignment)

   IF...THEN...END

   IF...THEN...ELSE...END

   CASE...OF...END

   CASE...OF...ELSE...END

   WHILE...DO...END

   EXIT

   RETURN

8. Other delimiter patterns are considered "operators," as follows:

   \...\ (caselabel designation "operator")

   [...] (partword and substring "operators")

   (...) (subexpression, array subscript, actual argument list, and function return value "operators")

   , (list item separation "operator")

9. Finally, implicit statement list brackets (associated with pairs of keywords such as THEN...ELSE and ELSE...END) are considered "operators," as are implicit statement separators between consecutive statements of the same statement list.

(The quotation marks flagging "operator" and "operand" as

technical terms are suppressed throughout the remainder of this section for readability.)

Basic Parameters

The five basic parameters of software science are determined in accordance with the criteria established for identfying operators and operands.

The theory defines four basic parameters pertaining to a program's implementation: the total operator count N_1, the total operand count N_2, the unique operator count η_1, and the unique operand count η_2. The total counts include all occurrences of operators/operands, while the unique counts disregard multiple occurrences of the same operator/operand. Although the issue of synonymy for operators has already been dealt with in the identification criteria, issues of synonymy for operands still remain. In particular, formal parameters are considered to be synonymous with corresponding actual arguments; therefore occurrences of formal parameter identifiers contribute to the total operand count but not to the unique operand count, with respect to the entire program. This rule is not, however, applied in the case of formal parameters passed by value and modified by the segment; because these are actually treated as special initialized-upon-entry local variables in the implementation language SIMPL-T, they are not considered to be synonymous operands with respect to the entire program.

The theory defines four additional basic parameters, analogous to those described above, but pertaining to an algorithm's or program's "shortest possible or most succinct form" (i.e., its "one-liner" functional specification, conceived as an assignment statement or procedure call

involving a single built-in routine). These are the total potential operator count N_1^*, the total potential operand count N_2^*, the unique potential operator count η_1^*, and the unique potential operand count η_2^*. (The modifier "potential" and the superscripted asterisk distinguish quantities pertaining to the functional specification from analogous quantities pertaining to the implementation.) The theory assumes, however, that the total potential operator/operand counts must always equal the unique potential operator/operand counts, because the most succinct specification would contain no redundant occurrences of operators/operands. An assumption is also made that the unique potential operator count is always equal to the constant 2, because

> "... the minimum possible number of operators ... must consist of one distinct operator for the name of the function or procedure and another to serve as an assignment or grouping symbol."

Thus, all software science quantities pertaining to a program's specification are completely determined (numerically speaking) by a single parameter, the unique potential operand count η_2^*. It is the fifth basic parameter of software science theory and is conceptually equivalent to the number of "logically distinct input/output parameters" for an algorithm or program. This count holds considerable significance in both the theory and its application, but unfortunately it is rather intractable for most nontrivial programs (i.e., those whose specifications are not easily stated as "one-liners" without gross oversimplification). For example, some logically distinct input parameters may appear as special constants embedded within the code, and the number of logically distinct output parameters represented within a printed report is often unclear.

An alternative and more tractable conceptualization defines this unique potential operand count simply as the number of distinct operands busy-on-entry (i.e., initially containing a value that is utilized or accessed by the algorithm or program) plus the number of distinct operands busy-on-exit (i.e., finally containing a value that was furnished by the algorithm or program to be utilized or accessed subsequently). For an individual segment, η_2^* may be estimated from the implementation by counting all of the global variables that are referenced, each of the formal parameters, one for both the input stream file and the output stream file (if they are read or written), and one for the function return value (if the segment is a function). It should be noted that this estimate is a lower bound since it disregards the possibility that a formal parameter which is passed by reference should be counted twice because it is both busy-on-entry and busy-on-exit. For an entire program, η_2^* may be estimated from the implementation by counting one for both the input stream file and the output stream file if the program reads or writes them, plus one for the set of control bits or option letters that might be used to regulate the program's execution.

Thus, for the programs examined in this study, the estimated number of unique potential operands is always either 2 or 3, depending on whether

```
output_listing := compile_and_execute( input_deck )
```

or

```
output_listing :=
     compile_and_execute( input_deck, option_letters )
```

states their functional specification. It is clear that this kind of estimation of η_2^* from the implementation is considerably more accurate for individual segments than for entire programs; this fact is partial motivation for the

particular estimation technique employed the second method of calculation discussed below.

Derived Properties

The derived properties of software science are defined in terms of the five basic parameters.

The vocabulary η is defined as

$$\eta = \eta_1 + \eta_2$$

and represents the cardinality of the set of logically distinct "symbols" used to implement the program. The length N is defined as

$$N = N_1 + N_2$$

and represents the abstract size of the program's implementation as measured in units of logically distinct "symbols." This property is closely associated with the number of syntactic tokens in the source code of a program; it can be considered a refinement of the rudimentary TOKENS aspect. The estimated length $\hat{N}$ is defined as

$$\hat{N} = (\eta_1 * \log_2(\eta_1)) + (\eta_2 * \log_2(\eta_2)) ,$$

reflecting one of the fundamental conjectures of the theory; namely, that the observed length of a program's implementation is a function solely of the number of unique operators/operands involved.

Considerable empirical evidence has supported the validity of this "length prediction equation" on the average (i.e., major software studies have reported correlation coefficients of between 0.95 and 0.99 for the relationship between N and $\hat{N}$ [Fitzsimmons & Love 78]). However, its accuracy for any given program may be low; the theory attributes this to the presence of so-called "impurities" indicating a lack of polish in a program. These impurities include instances of unnecessary redundancy and needless

constructions, such as inverse operations that cancel each other, common subexpressions, or unreachable statements. This has led some researchers to view the discrepancy between N and $\hat{N}$ as a possible software quality measure. For these reasons, it was desirable to examine the %DIFFERENCE(N,$\hat{N}$) aspect, calculated as

$$(|\hat{N} - N|) / (N) ,$$

which normalizes the degree of discrepancy.

The volume V is defined as

$$V = N * \log_2(\eta)$$

and represents the abstract size of the program's implementation as measured in units of information-theoretic bits. Specifically, it is the minimum number of bits required to encode the implementation as a sequence of fixed-width binary strings (since it is the product of the total number of "symbols" and the minimum bandwidth required to distinguish each of the unique "symbols"). The potential volume V^* is defined analogously as

$$\begin{aligned} V^* &= N^* * \log_2(\eta^*) \\ &= (N_1^* + N_2^*) * \log_2(\eta_1^* + \eta_2^*) \\ &= (\eta_1^* + \eta_2^*) * \log_2(\eta_1^* + \eta_2^*) \\ &= (2 + \eta_2^*) * \log_2(2 + \eta_2^*) . \end{aligned}$$

The potential volume of any algorithm or program is theoretically independent of any language in which it might be implemented; thus,

> "Provided that η_2^* is evaluated as the number of conceptually unique operands involved, V^* appears to be a most useful measure of an algorithm's content."

The program level L is defined as

$$L = V^* / V$$

and, as a ratio of volumes, can only take on values between zero and one; it quantifies the intuitive concept of "level of abstraction" for an implementation. Since the potential

volume of any given algorithm is constant, the formula indicates an inverse relationship (as desired intuitively) between level of abstraction (measured by L) and size (measured by V). The theory also attaches meaning to the reciprocal of program level, defining <u>difficulty</u> D as

$$D = 1 / L$$

which may alternatively be viewed as the amount of redundancy within an implementation.

Unfortunately, this definition of program level is not particularly useful since it is difficult (as discussed above) to determine exactly a program's unique potential operand count η_2^* or its potential volume V* . Desiring to be able to measure program level even if these quantities were unavailable, Halstead conjectured that an <u>estimated program level</u> $\hat{L}$, defined as

$$\hat{L} = (\eta_1^* / \eta_1) * (\eta_2 / N_2)$$
$$= (2 / \quad) * (\quad / N) ,$$

could be measured directly from an implementation alone, without a specification or the unique potential operand count η_2^* . With only limited evidence supporting the validity of this estimate, the theory makes the qualified claim that

> "... for many purposes L and $\hat{L}$ may be used interchangeably to specify the level at which a program has been implemented, at least for smaller programs."

Although most software science studies (e.g., [Elshoff 76; Love & Bowman 76; Curtis et al. 79]) have had no choice but to rely upon this "program level prediction equation" to calculate the derived properties, the program level estimator $\hat{L}$ has been criticized publically [Oldehoeft 77] for unsatisfactory behavior under certain conditions. The questionable validity of this prediction equation is the principal motivation for considering the two alternative methods of calculation discussed below.

In any event, the theory employs $\hat{C}$ internally, defining <u>intelligence content</u> I as

$$I = \hat{C} * V$$

and proposing it as a measure of "how much is said" in an algorithm or program (i.e., its information content). This intelligence content quantity represents the amount of detail expressed in an implementation but weighted by its level of expression. By definition, I is determinable from an implementaion alone. If there is a strong relationship between L and $\hat{C}$, intelligence content I would be approximately equal to potential volume V^*. In fact, Halstead originally demonstrated that the removal of program "impurities" (as described above) consistently improved the numerical agreement between V^* and I. Normalizing the degree of discrepancy between these two quantities, the %DIFFERENCE(V*,I) aspect, calculated as

$$(|I - V^*|) / (V^*) ,$$

may be interpreted as another possible software quality measure, according to the theory.

The <u>language level</u> λ is defined as

$$\lambda = L * V^*$$
$$= L * L * V$$
$$= V^* * V^* / V$$

and claims to quantify the popular intuitive concept known by the same name. The theory suggests that λ should remain relatively constant for any particular implementation language while the implemented algorithm itself is allowed to vary. Empirical evidence from a carefully constructed set of programs, each implemented in several common programming languages, indicated that the ordering of mean values for λ (which ranged from about 0.8 for assembly language to about 1.6 for PL/1) concurred exactly with the generally accepted intuitive ordering of the languages themselves.

The <u>effort</u> E is defined as

$$E = V * D$$
$$= V / L$$
$$= V * V / V'$$

but this quantity does not purport to measure development effort in the usual sense. Rather, the theory originally restricted

> "... the concept of programming effort to be the mental activity required to reduce a preconceived algorithm to an actual implementation in a language in which the implementor (writer) is fluent ..."

According to further elaboration of the theory [Gordon 79], this property represents the effort required (under ideal conditions) to comprehend an implementation rather than to produce it; E may thus be interpreted as a measure of program clarity. The effort property is considered to have the dimension either of bits or of "elementary mental discriminations." Borrowing from research in psychology, the theory converts this amount of mental effort into an externally observable duration of time, defining the <u>estimated time</u> T as

$$T = E / S$$

where S is the so-called Stroud rate, i.e., the number of "elementary mental discriminations" made by a programmer (comprehender) per second. Psychologists had shown that $5 \leq S \leq 20$ and Halstead determined empirically that $S = 18$ was a reasonable value.

Finally, the theory purports to quantify one other externally observable property, namely, the total number of "delivered" bugs in an implementation. The <u>estimated bugs</u> B property is defined as

$$B = L * E / E_0$$
$$= V / E_0$$

where E_0 is defined as "the mean number of elementary

mental discriminations between potential errors in programming." The theory argues that E_o = 3000 is a reasonable value. This number of bugs may be interpreted as either the expected number of errors remaining in a delivered program or the number of errors observed during program testing; both interpretations have received some empirical support.

Calculation Methods

Because the validity of the "program level prediction equation" is suspect (as discussed above), this study employed two different methods for calculating software science quantities: one relies directly upon this estimate, the other does not.

Both methods calculate exact values for some derived properties via their defining formulas directly from the implementation's basic parameters. The methods are therefore identical with regard to the following measured aspects: VOCABULARY, LENGTH, ESTIMATED LENGTH, %DIFFERENCE(N,$\hat{N}$), VOLUME, INTELLIGENCE CONTENT, and ESTIMATED BUGS. But, because reasonable values for unique potential operand counts are generally unavailable (from either the specification or the implementation) for programs of the size considered in this study, both methods of calculation can only approximate the remaining derived properties by relying upon various estimates. Due to the intrinsically high degree of interrelationship among the software science quantities, it generally suffices to approximate just one additional derived property via some estimation technique; the remaining derived properties can then all be approximated in turn via their defining formulas from the known exact values plus the estimated value. The methods therefore differ in their choice of quantity to be

estimated, in their estimation technique, and with regard to the following measured aspects: PROGRAM LEVEL, DIFFICULTY, POTENTIAL VOLUME, %DIFFERENCE(V*,I), LANGUAGE LEVEL, EFFORT, and ESTIMATED TIME.

The first method relies upon a "theoretical" estimation of the program level quantity. The estimated program level $\hat{L}$ is calculated directly from the program's implementation via its defining formula and then substituted as an approximation for the (true) program level L. Under this method the exact value for intelligence content I is, by definition, always equal to the approximate value for potential volume V^*; hence it is pointless to examine the %DIFFERENCE(V*,I) aspect under this method of calculation.

The second method relies upon an "empirical" estimation of the language level quantity. A program's language level is approximated as the mean value of estimates for the language levels of the segments comprising the program. An estimate of each segment's language level can be calculated directly from the implementation (via the defining formulas for λ, V^*, and V), using an estimate of the segment's unique potential operand count η_2^* in addition to the exact values of the segment's other basic parameters N_1, N_2, η_1, and η_2. The unique potential operand estimate is obtained by counting operands that are busy-on-entry or busy-on-exit (as discussed above); this technique seems quite reasonable when applied to segments, most of which are small enough. Use of the mean estimate for λ across the individual segments of an entire program was inspired by the experimental treatment of language level given in Halstead's book. Under this method of calculation, all of the derived properties defined above are distinct and nonredundantly calculated.

CHAPTER VI

VI. INVESTIGATIVE METHODOLOGY

This chapter describes the steps taken to guide the planning, execution, and analysis of the experimental investigation reported in this dissertation. The investigative methodology outlined here was devised as a vehicle for research in software engineering. It relies upon established principles and techniques for scientific research: empirical study, controlled experimentation, and statistical analysis.

The central feature of the investigative methodology is a "differentiation-among-groups-by-aspects" paradigm. The research goal is to answer the question, what differences exist among the treatment groups (which represent different programming environments) as indicated by differences on measured aspects (which reflect quantitative characteristics of software phenomena)? This use of "difference discrimination" as the analytical technique dictates a statistical model of homogeneity hypothesis testing that influences nearly every element of the investigative methodology.

Other analytical techniques could have been employed:

estimation of the magnitude of differences between experimental treatments,

correlations between measured aspects across all experimental treatments,

multivariate analysis (rather than multiple univariate analyses in parallel, as is the case here), or

factor analysis (breakdown of variance in one aspect among the other measured aspects),

to name a few examples. These are useful techniques and may be used at a later time to answer other research questions.

CHAPTER VI

For the present investigation, difference discrimination was chosen as a reasonable "first-cut" probe of the empirical data collected for the research project; by taking this conservative approach, information may be obtained to help guide more refined probes in the future.

Although the methodology is built around running an experiment, collecting data, and making statistical tests, these activities (i.e., the execution phase) play a small role within the overall investigative methodology, in comparison to the planning and analysis phases. This is readily apparent from the schematic in Figure 2, which charts some of the relationships among the various elements (or steps) of the investigative methodology. Another feature of the investigative methodology is the careful distinction made during the analysis phase between objective results (the empirical scores for the metrics and the statistical conclusions they infer) and subjective results (interpretations of the objective results in light of intuition, research goals, etc.).

The remainder of this chapter outlines the overall method by defining each step and discussing how it was applied. Further details of certain steps are given within other chapters of the dissertation, as follows:

Step 5	Research Frameworks	Chapter VIII
Step 6	Experimental Design	Chapter III
Step 7	Collected Data	Chapter III
Step 10	Statistical Conclusions	Chapter VII
Step 11	Research Interpretations	Chapter VIII

Step 1: Questions of Interest

Several questions of interest were initiated and refined so that answers might be given in the form of

FIGURE 2

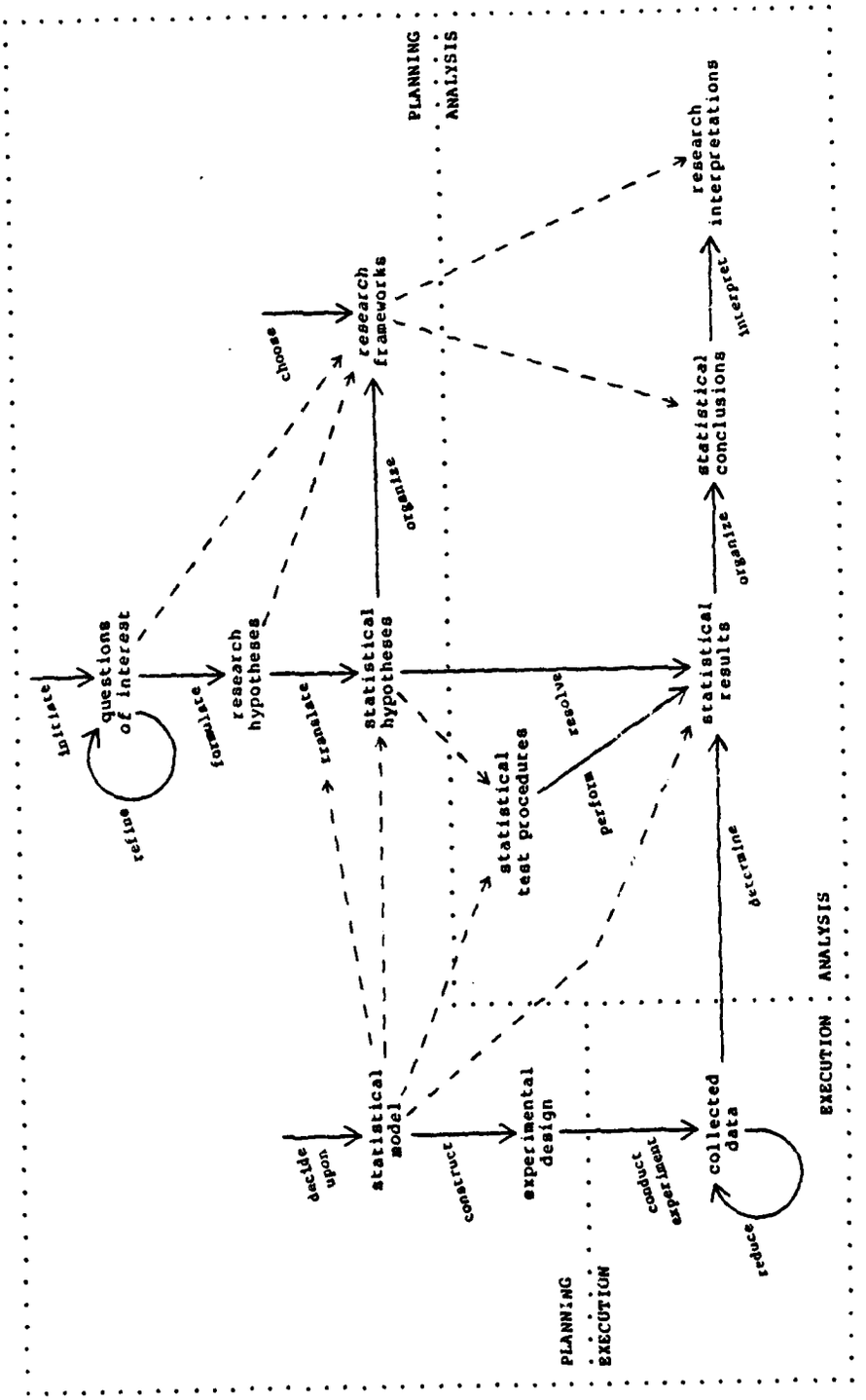


Figure 2. Investigative Methodology Schematic

statistical conclusions and research interpretations. Questions were formulated on the basis of several concerns: (1) software development rather than software maintenance, (2) a desire to assess the effectiveness of disciplined team programming, in comparison to undisciplined team programming and individual programming, (3) quantitatively measurable aspects of the process and product, and (4) the analytical technique of difference discrimination. The questions of interest took the final form, "During software development, what comparisons between the effects of the three programming environments,

(a) individual programming under an ad hoc approach,

(b) team programming under an ad hoc approach,

(c) team programming under a disciplined methodology,

appear as differences in quantitatively measurable aspects of the software development process and product? Furthermore, what kind of differences are exhibited and what is the direction of these differences?"

Step 2: Research Hypotheses

Since the investigative methodology involves hypothesis testing, it is necessary to have fairly precise statements, called research hypotheses, which are to be either supported or refuted by the evidence. The second step in the method was to formulate these research hypotheses, disjoint pairs designated null and alternative, from the questions of interest.

A precise meaning was given to the notion of "difference." The investigation considered both (a) differences in central tendency or average value, and (b) differences in variability around the central tendency, of observed values of the quantifiable programming aspects. It should be noted that this decision to examine both location

and dispersion comparisons among the experimental groups brought a pervasive duality to the entire investigation (i.e., two sets of statistical tests, two sets of statistical results, two sets of conclusions, etc.--always in parallel and independent of each other), since it addresses both the expectancy and the predictability of behavior under the experimental treatments.

Some vagueness was removed regarding the size of the particular programming task by making explicit the implicit restriction that completion of the task not be beyond the capability of a single programmer working alone for a reasonable period of time. Additionally, a large set of programming aspects were specified; they are discussed in Chapters IV and V. For each programming aspect there were similar questions of interest, similar research hypotheses and similar experiments conducted in parallel.

The schema for the research hypotheses may be stated as "In the context of a one-person-do-able software development project, there < is not | is > a difference in the < location | dispersion > of the measurements on programming aspect < X > between individuals (AI), ad hoc teams (AT), and disciplined teams (DT)." For each programming aspect 'X' in the set under consideration, this schema generates two pairs of nondirectional research hypotheses, depending upon the selection of 'is not' or 'is' corresponding to the null and alternative hypothesis, and the selection of 'location' or 'dispersion' corresponding to the type of difference.

Step 3: Statistical Model

The choice of a statistical model makes explicit various assumptions regarding the experimental design, the

dependent variables, the underlying population distributions, etc. Because the study involves a homogeneity-of-populations problem with shift and spread alternatives, the multi-sample model used here requires the following: independent populations, independent and random sampling within each population, continuous underlying distributions for each population, homoscedasticity (equal variances) of underlying distributions, and interval scale of measurement [Conover 71, pp. 65-67] for each programming aspect. Although random sampling was not explicitly achieved in this study by rigorous sampling procedures, it was nonetheless assumed on the basis of the apparent representativeness of the subject pool and the lack of obvious reasons to doubt otherwise. Due to the small sample sizes, the unknown shape of the underlying distributions, and the partially exploratory nature of the study, a nonparametric statistical model was used.

Whenever statistics is employed to "prove" that some systematic effect--in this case, a difference among the groups--exists, it is important to measure the risk of error. This is usually done by reporting a significance level α [Conover 71, p. 79], which represents the probability of deciding that a systematic effect exists when in fact it does not. In the model, the hypothesis testing for each programming aspect was regarded as a separate independent experiment. Consequently, the significance level is controlled and reported experimentwise (i.e., per aspect). While the assumption of independence between such experiments is not entirely supportable, this procedure is valid as long as statistical inferences that couple two or more of the programming aspects are avoided or properly qualified. In this study, statements regarding interrelationships among aspects are made only within the interpretations in Chapter VIII.

CHAPTER VI

Step 4: Statistical Hypotheses

The research hypotheses must be translated into statistically tractable form, called statistical hypotheses. A correspondence, governed by the statistical model, exists between application-oriented notions in the research hypotheses (e.g., typical performance of a programming team under the disciplined methodology) and mathematical notions in the statistical hypotheses (e.g., expected value of a random variable defined over the population from which the disciplined teams are a representative sample). Generally speaking, only certain mathematical statements involving pairs of populations are statistically tractable, in the sense that standard statistical procedures are applicable. Statements that are not directly tractable may be decomposed into tractable (sub)components whose results are properly recombined after having been decided individually.

In this study, the research hypotheses are concerned with directional differences among three programming environments. Since the corresponding mathematical statements are not directly tractable, they were decomposed into the set of seven statistical hypotheses pairs shown below. As a shorthand notation for longer English sentences, symbolic "equations" are used to express these statistical hypotheses. The - symbol denotes negation. The + symbol denotes pooling. The = , ≠ , and < symbols indicate comparisons on the basis of either the location or dispersion of the dependent variables.

The hypotheses pair

null:	alternative:
AI = AT = DT	-(AI = AT = DT)

addresses the existence of an overall difference among the groups. However, due to the weak nondirectional

alternative, it cannot indicate which groups are different or in what direction a difference lies. Standard statistical practice prescribes that a successful test for overall difference among three or more groups be followed by tests for pairwise differences. The hypotheses pairs

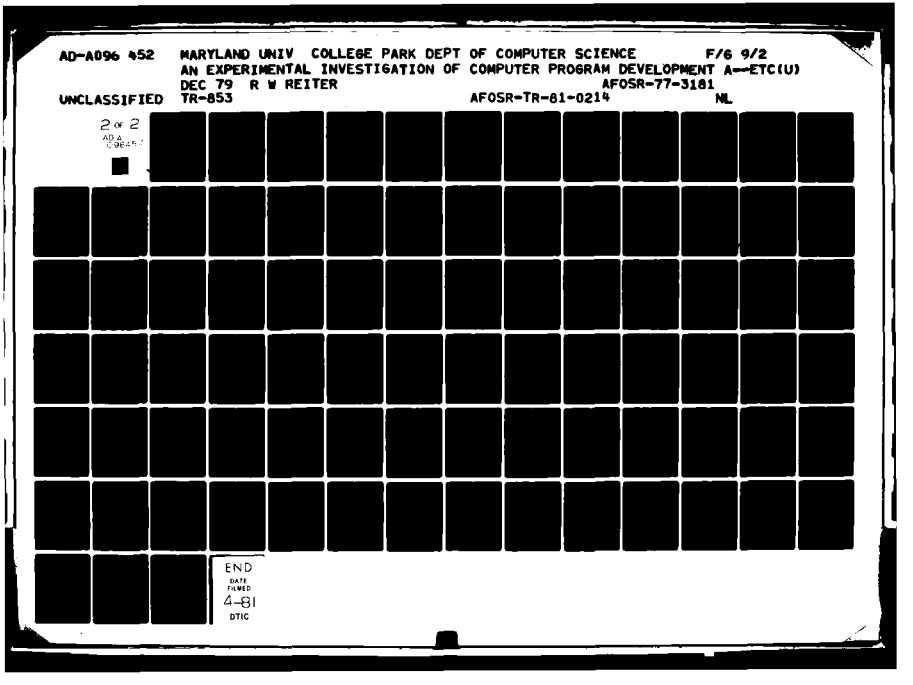
null:	alternative:		
AI = AT	AI ≠ AT	or AI < AT	or AT < AI
AT = DT	AT ≠ DT	or AT < DT	or DT < AT
AI = DT	AI ≠ DT	or AI < DT	or DT < AI

address the existence and direction of pairwise differences between groups. The results of these pairwise comparisions were used to refine the overall comparison. Data collected for a set of experiments may often be legitimately reused to "simulate" other closely related experiments, by combining certain samples together and ignoring the original distinction(s) between them. It is meaningful, in the context of this study's experimental design, to compare any two groups pooled against the third since (1) AI and AT are both undisciplined, while DT is disciplined; (2) AT and DT are both teams, and AI is individuals; and (3) under the assumption that disciplined teams behave like individuals--which is part of the study's basic premise, DT and AI can be pooled and compared with AT acting as a control group. The hypotheses pairs

null:	alternative:		
AI+AT = DT	AI+AT ≠ DT	or AI+AT < DT	or DT < AI+AT
AT+DT = AI	AT+DT ≠ AI	or AT+DT < AI	or AI < AT+DT
AI+DT = AT	AI+DT ≠ AT	or AI+DT < AT	or AT < AI+DT

address the existence and direction of such pooled differences. The results of these pooled comparisons were used to corroborate the overall and pairwise comparisons.

Thus, for each programming aspect, the research hypotheses pair corresponds to seven different pairs (null and alternative) of statistical hypotheses. The results of


AD-A096 452 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
AN EXPERIMENTAL INVESTIGATION OF COMPUTER PROGRAM DEVELOPMENT A--ETC(U)
DEC 79 R W REITER AFOSR-77-3181
UNCLASSIFIED TR-853 AFOSR-TR-81-0214 NL
2 OF 2
AD A
END
DATE FILMED
4-81
DTIC

testing each set of seven hypotheses must be abstracted and organized into one statistical conclusion using the first research framework discussed in the next step.

Step 5: Research Frameworks

The research frameworks provide the necessary organizational basis for abstracting and conceptualizing the massive volume of statistical hypotheses (and statistical results that follow) into a smaller and more intellectually manageable set of conclusions. Three separate research frameworks have been chosen: (1) the framework of possible overall comparison outcomes for a given programming aspect, (2) the framework of dependencies and intuitive relationships among the various programming aspects considered, and (3) the framework of basic suppositions regarding expected effects of the experimental treatments on the comparison outcomes for the entire set of programming aspects. The first framework is employed in the statistical conclusions step because it can be applied in a statistically tractable manner, while the remaining two frameworks are reserved for employment in the research interpretations step since they are not statistically tractable and involve subjective judgement.

Since a finite set of three different programming environments (AI, AT, and DT) are being compared, there exists the following finite set of thirteen possible overall comparison outcomes for each aspect considered:

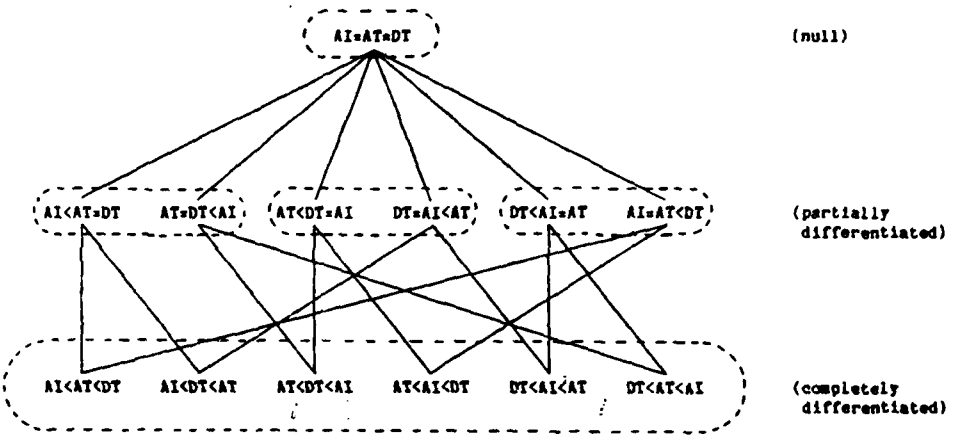
$$
\begin{array}{l}
AI = AT = DT \\
\left.\begin{array}{l} AI < AT = DT \\ AT = DT < AI \end{array}\right\} AI \neq AT = DT \\
\left.\begin{array}{l} AT < DT = AI \\ DT = AI < AT \end{array}\right\} AT \neq DT = AI \\
\left.\begin{array}{l} DT < AI = AT \\ AI = AT < DT \end{array}\right\} DT \neq AI = AT
\end{array}
\qquad
\left.\begin{array}{l}
AI < AT < DT \\ AI < DT < AT \\ AT < DT < AI \\ AT < AI < DT \\ DT < AI < AT \\ DT < AT < AI
\end{array}\right\} AI \neq AT \neq DT
$$

There is a hierarchical lattice of increasing separation and directionality among these possible overall comparison outcomes as shown in Figure 3. These thirteen possible overall comparison outcomes comprise the first research framework and may be viewed as providing a complete "answer space" for the questions of interest. It is clear that any consistent set of two-way comparisons (such as represented in the statistical hypotheses or statistical results) may be associated with a unique one of these three-way comparisons. This framework is the basis for organizing and condensing the seven statistical results into one statistical conclusion for each programming aspect considered.

Since a large set of interrelated programming aspects are being examined, it would be desirable to summarize many of the "per aspect" hypotheses and results into statements which refer to several aspects simultaneously. For example, average number of statements per segment is one aspect directly dependent on two other aspects: number of segments and number of statements. Other interrelationships are more intuitive, less tractable, or only suspected, for example, the "trade-off" between global variables and formal parameters. A simple classification of the programming aspects into groups of intuitively related aspects at least provides a framework for jointly interpreting the corresponding statistical conclusions in light of the underlying issues by which the aspects themselves are related. The programming aspects considered in this study

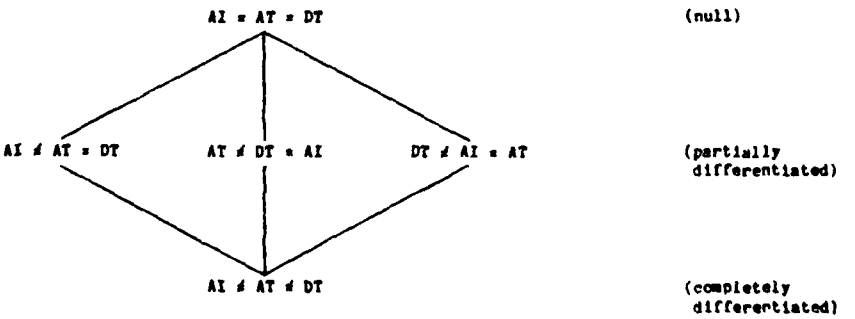
FIGURE 3

Figure 3.1 Lattice of Possible Directional Outcomes for Three-way Comparison



N.B. The circles indicate which directional outcomes correspond to the same nondirectional outcome.

Figure 3.2 Lattice of Possible Nondirectional Outcomes for Three-way Comparison



were classified according to a particular set of nine higher-level programming issues (such as data variable organization, for example); details are given in Chapter VIII. This second research framework is the basis for abstracting and interpreting what the study's findings indicate about these higher-level programming issues, as well as explicitly mentioning several individual relationships among the programming aspects and their conclusions.

Since the design of the experiments, the choice of treatments, etc., were at least partially motivated by certain general beliefs regarding software development (e.g., "disciplined methodology reduces software development costs"), it should be possible to explicitly state what comparison outcomes among the experimental treatments were expected a priori for which programming aspects. A list of preplanned expectations (so-called "basic suppositions") for the outcomes of each aspect's experiment would provide a framework for evaluating how well the experimental findings as a whole support the underlying general beliefs (by comparing the actual outcomes with the basic suppositions across all the programming aspects). Such a list of basic suppositions was conceived prior to conducting the experiments, and it constitutes the third research framework; details are given in Chapter VIII. This framework is the basis for interpreting the study's findings as evidence in favor of the basic suppositions and general beliefs.

Step 6: Experimental Design

The experimental design is the plan according to which the experiment is actually executed. It is based upon the statistical model and deals with practical issues such as

experimental units, treatments, local control, etc. The experimental design employed for this study has been discussed in considerable detail in Chapter III.

Step 7: Collected Data

The pertinent data to carry out the experimental design was collected and processed to yield the information to which the statistical test procedures were applied. Some details of these activities have been given in Chapter III.

Step 8: Statistical Test Procedures

A statistical test procedure is a decision mechanism, founded upon general principles of mathematical probability and combinatorics and upon a specific statistical model (i.e., requiring certain assumptions), which is used to convert the statistical hypotheses together with the collected data into the statistical results. As dictated by the statistical model, the statistical tests used in the study were nonparametric tests of homogeneity of populations against shift alternatives for small samples. Nonparametric tests are slightly more conservative (in rejecting the null hypothesis) than their parametric counterparts; nonparametric tests generally use the ordinal ranks associated with a linear ordering of a set of scores, rather than the scores themselves, in their computational formulas. In particular, the standard Kruskal-Wallis H-test [Siegel 56, pp. 184-193] and Mann-Whitney U-test [Siegel 56, pp. 116-127] were employed in the statistical results step. Ryan's Method of Adjusted Significance Levels [Kirk 68, p. 97, pp. 495-497], a standard procedure for controlling the experimentwise significance level when several tests are performed on the same scores as one experiment, was also employed in the statistical conclusions step.

CHAPTER VI

The Kruskal-wallis test is used in three-sample situations to test an X = Y = Z null hypothesis; its test statistic is computed as

$$H = 12*[(Rx^2/nx)+(Ry^2/ny)+(Rz^2/nz)]/[n*(n+1)] - 3*(n+1)$$

where Rx , Ry , and Rz are the respective sums of the ranks for scores from the X, Y, and Z samples; n equals nx+ny+nz where nx , ny , and nz are the respective sample sizes. The Mann-Whitney test is used in two-sample situations to test an X = Y null hypothesis; its test statistic is computed as

$$U = \min[\ nx*ny + nx*(nx+1)/2 - Rx\ ,\ ny*nx + ny*(ny+1)/2 - Ry\ ]$$

where Rx , Ry , nx , and ny are defined as before.

For every statistical test, there exists a one-to-one mapping, usually given in statistical tables, between the test statistic--a value completely determined by the sample data scores--and the critical level. The critical level [Conover 71, p. 81] is defined as the minimum significance level at which the statistical test procedure would allow the null hypothesis to be rejected (in favor of the alternative) for the given sample data. Thus critical level represents a concise standarized way to state the full result of any statistical test procedure. Two-tailed rejection regions are applied for tests involving nondirectional alternative hypotheses, and one-tailed rejection regions are applied for tests involving directional alternative hypotheses, so that the stated critical level always pertains directly to the stated alternative hypothesis. A decision to reject the null hypothesis and accept the alternative is mandated if the critical level is low enough to be tolerated; otherwise a decision to retain the null hypothesis is made.

The Ryan's procedure is used in situations involving multiple pairwise comparisons, in order to properly account

for the fact that each pairwise test is made in conjunction with the others, using the same sample data. The individual critical levels $\hat{\alpha}$ obtained for each pairwise test in isolation are adjusted to proper experimentwise critical levels $\hat{\alpha}'$ via the formula

$$\hat{\alpha}' = [(r+1)*k/2] * \hat{\alpha}$$

where k is the total number of samples; and r is the number of (other) samples whose rank means fall between the rank means of the particular pair of samples being compared. A simple "minimax" step--taking the maximum of the several adjusted pairwise critical levels, plus the overall comparison critical level, which are all minimum significance levels--completes the procedure, yielding a single critical level associated jointly with the overall and pairwise comparisons.

These tests and procedures apply straightforwardly when differences in location are considered. A slight modification makes them applicable for differences in dispersion: prior to ranking, each score value is simply replaced by its absolute deviation from the corresponding within-group sample median [Nemenyi et al. 77, pp. 266-270]. It should be noted that this modification results in only an approximate method for solving a tough statistical problem, namely, testing whether one population is more variable than another [Nemenyi et al. 77, pp. 279-283]. The modification is not statistically valid in the general case (it weakens the power of the test procedures and can yield inaccurate critical levels when testing for dispersion differences), but every other available method also has serious limitations. This method has been shown (empirically via Monte Carlo techniques) to possess reasonable accuracy, as long as the underlying distributions are fairly symmetrical, and is readily adapted to the study's three-way comparison situation.

CHAPTER VI

Step 9: Statistical Results

A statistical result is essentially a decision reached by applying a statistical test procedure to the set of collected and refined data, regarding which one of the corresponding pair (null, alternative) of statistical hypotheses is indeed supported by that data. For each pair of statistical hypotheses, there is one statistical result consisting of four components: (1) the null hypothesis itself; (2) the alternative hypothesis itself; (3) the critical level, stated as a probability value between 0 and 1; and (4) a decision either to retain the null hypothesis or to reject it in favor of (i.e., accept) the alternative hypothesis.

By convention, the null hypothesis is that no systematic difference appears to exist, and the alternative hypothesis purports that some systematic difference exists. The critical level is associated with erroneously accepting the alternative hypothesis (i.e., claiming a systematic difference when none in fact exits). The decision to retain or reject is reached on the basis of some tolerable level of significance, with which the critical level is compared to see if it is low enough. In cases where a null hypothesis is rejected, the appropriate directional alternative hypothesis (if any) is used to indicate the direction of the systematic difference, as determined by direct observation from the sample medians in conjunction with a one-tailed test.

Conventional practice is to fix an arbitrary significance level (e.g., 0.05 or 0.01) in advance, to be used as the tolerable level; critical levels then serve only as stepping-stones toward reaching decisions and are not reported. For this partially exploratory study, it was

deemed more appropriate to fix a tolerable level only for the purpose of a screening decision (simply to purge those results with intolerably high critical levels) and to explicitly attach any surviving critical level to each statistical result. This unconventional practice yields statistical results in a more meaningful and flexible form, since the significance or error risk of each result may be assessed individually, and results at other more stringent significance levels may be easily determined. Furthermore, the necessary information is retained for properly recombining multiple related results on an experimentwise basis in the statistical conclusions step.

The tolerable level of significance used throughout this study to screen critical levels was fixed at under 0.20. Although fairly high for a confirmatory study, it is reasonable for a partially exploratory study, such as this one, seeking to discover even slight trends in the data. A critical level of 0.20 means that the odds of obtaining test scores exhibiting the same degree of difference, due to random chance fluctuations alone, are one in five.

As an example, the seven statistical results for location comparisons on the programming aspect STATEMENT TYPE COUNTS\IF are shown below. (N.B. The asterisks will be explained in Step 10.)

null hypothesis	alternative hypothesis	critical level	(screening) decision	
AI = AT = DT	-(AI = AT = DT)	.063	reject	
AI = AT	AI < AT	.046	reject	
AI = DT	AI ≠ DT	>.999	retain	
AT = DT	DT < AT	.011	reject	
AI+AT = DT	DT < AI+AT	.088	reject	*
AI+DT = AT	AI+DT < AT	.009	reject	
AT+DT = AI	AT+DT ≠ AI	.335	retain	*

Observe that the stated decisions simply reflect the application of the 0.20 tolerable level to the stated critical levels. Results under more stringent levels of significance can be easily determined by simply applying a

lower tolerable level to form the decisions; e.g., at the 0.05 significance level, only the AI < AT, DT < AT, and AI+DT < AT alternative hypotheses would be accepted; only the AI+DT < AT hypothesis would be accepted at the 0.01 level.

Step 10: Statistical Conclusions

The volume of statistical results are organized and condensed into statistical conclusions according to the prearranged research framework(s). A statistical conclusion is an abstraction of several statistical results, but it retains the same statistical character, having been derived via statistically tractable methods and possessing an associated critical level.

The first research framework mentioned above was employed to reduce the seven statistical results (with seven individual critical levels) for each programming aspect to a single statistical conclusion (with one overall critical level) for that aspect. The statement portion of a statistical conclusion is simply one of the thirteen possible overall comparison outcomes. Each overall comparison outcome is associated with a particular set of statistical results whose outcomes support the overall comparison outcome in a natural way. For example (reading from the fifth row of the chart in Figure 4), the DT = AI < AT conclusion is associated with the following results:

reject AI = AT = DT in favor of -(AI = AT = DT),
reject AI = AT in favor of AI < AT,
retain AI = DT,
reject AT = DT in favor of DT < AT, and
reject AI+DT = AT in favor of AI+DT < AT.

Since the other two comparisons (AI+AT versus AT, AT+DT

versus AI) are in a sense orthogonal to the overall comparison outcome (DT = AI < AT), their results are considered irrelevant to this conclusion. The chart in Figure 4 shows exactly which results are associated with each conclusion: the relevant comparisons, the null hypotheses to be retained, and the alternative hypotheses to be accepted. The other portion of a statistical conclusion is the critical level associated with erroneously accepting the statement portion. It is computed from the individual critical levels of certain germane results.

A simple algorithm based on the chart in Figure 4 was used to generate the statistical conclusions (and compute the overall critical level) automatically from the statistical results. For each programming aspect, the algorithm compared the set of actual results obtained for the seven statistical hypotheses pairs to the set of results associated (in the chart) with each conclusion, searching for a match. Ryan's procedure was used to properly combine the individual critical levels for the overall result and the relevant pairwise results, by adjusting them via the formula and then taking their maximum. The critical levels for the relevant pooled results were then factored, by a simple formula based on the multiplicative rule for the joint probability of independent events.

Continuing the example started in Step 9, the statistical results shown there for location comparisons on the STATEMENT TYPE COUNTS\IF aspect are reduced to the statistical conclusion DT = AI < AT with .078 critical level overall. The five results not marked with an asterisk in Step 9 match the five results associated above with the DT = AI < AT outcome. (Note that the other two marked results represent comparisons that are irrelevant to this conclusion.) The .046 and .011 critical levels for the two

FIGURE 4

Figure 4. Association Chart for Results and Conclusions

The following chart specifies the set of statistical results associated with each statistical conclusion. An asterisk indicates a comparison not relevant to that conclusion; the null hypothesis appears wherever it must be retained; and the alternative hypothesis to be accepted appears wherever the null hypothesis must be rejected. If a set of results does not satisfy the criteria associated with some non-null conclusion (i.e., does not match on all relevant comparisons listed in some row), then it defaults to the null conclusion.

conclusions:	results: AI=AT=DT	AI=AT	AI=DT	AT=DT	AI+AT=DT	AI+DT=AT	AT+DT=AI
AI = AT = DT	[default]						
AI < AT = DT	-(AI=AT=DT)	AI<AT	AI<DT	AT=DT	*	*	AI<AT+DT
AT = DT < AI	-(AI=AT=DT)	AT<AI	DT<AI	AT=DT	*	*	AT+DT<AI
AT < DT = AI	-(AI=AT=DT)	AT<AI	AI=DT	AT<DT	*	AT<AI+DT	*
DT = AI < AT	-(AI=AT=DT)	AI<AT	AI=DT	DT<AT	*	AI+DT<AT	*
DT < AI = AT	-(AI=AT=DT)	AI=AT	DT<AI	DT<AT	DT<AI+AT	*	*
AI = AT < DT	-(AI=AT=DT)	AI=AT	AI<DT	AT<DT	AI+AT<DT	*	*
AI < AT < DT	-(AI=AT=DT)	AI<AT	AI<DT	AT<DT	AI+AT<DT	*	AI<AT+DT
AI < DT < AT	-(AI=AT=DT)	AI<AT	AI<DT	DT<AT	*	AI+DT<AT	AI<AT+DT
AT < DT < AI	-(AI=AT=DT)	AT<AI	DT<AI	AT<DT	*	AT<AI+DT	AT+DT<AI
AT < AI < DT	-(AI=AT=DT)	AT<AI	AI<DT	AT<DT	AI+AT<DT	AT<AI+DT	*
DT < AI < AT	-(AI=AT=DT)	AI<AT	DT<AI	DT<AT	DT<AI+AT	AI+DT<AT	*
DT < AT < AI	-(AI=AT=DT)	AT<AI	DT<AI	DT<AT	DT<AI+AT	*	AT+DT<AI

FIGURE 4

pairwise differences are adjusted to .070 and .033, respectively, and the maximum among those adjusted values and the .063 overall difference critical level is .070. The relevant pooled comparison critical level of .008 is factored in by taking the complement of the products of the complements:

$$1 - [(1 - .069)*(1 - .008)] = .078$$

Thus, the statistical conclusions are in one-to-one correspondence with the research hypotheses and provide concise answers on a "per aspect" basis to the questions of interest. Further details and complete listing of the statistical conclusions for this study are presented in Chapter VII.

Step 11: Research Interpretations

The final step in the method is to interpret the statistical conclusions in view of any remaining research framework(s), the researcher's intuitive understanding, and the work of other researchers. These research interpretations provide the opportunity to augment the objective findings of the study with the researcher's own professional judgment and insight. The second and third research frameworks mentioned above--namely, the intuitive relationships among the various programming aspects and the basic suppositions governing their expected outcomes--were considered important for this purpose. However these particular research frameworks can only be utilized for the research interpretations, since they are not amenable to rigorous manipulation. Nonetheless, within these frameworks based upon intuitions about the software metrics and programming environments under consideration, the study bears some of its most interesting results and implications. Complete details and discussion of the research

interpretations of this study appear in Chapter VIII.

CHAPTER VII

VII. OBJECTIVE RESULTS

This chapter reports the objective results of the study, namely, the statistical conclusions for each programming aspect considered. In keeping with the empirical and statistical character of these conclusions, the tone of discussion here is purposely somewhat disinterested and analytical. All interpretive discussion is deferred to Chapter VIII, in accordance with the investigative methodology.

Each statistical conclusion is expressed in the concise form of a three-way comparison outcome "equation." It states any observed differences, and the directions thereof, among the programming environments represented by the three groups examined in the study: ad hoc individuals (AI), ad hoc teams (AT), and disciplined teams (DT). The equality AI = AT = DT expresses the null outcome that there is no systematic difference among the groups. An inequality, e.g., AI < AT = DT or DT < AI < AT, expresses a non-null (or alternative) outcome that there are certain systematic difference(s) among the groups in stated direction(s). A critical level value is also associated with each non-null (or alternative) outcome, indicating its individual reliability. This value is the probability of having erroneously rejected the null conclusion in favor of the alternative; it also provides a relative index of how pronounced the differences were in the sample data.

The remainder of this chapter consists of (a) presenting the full set of conclusions, (b) evaluating their impact as a whole, (c) exposing a "relaxed differentiation" view of the conclusions, (d) exposing a "directionless" view of the conclusions, and (e) individually highlighting a few

of the more noteworthy conclusions.

Presentation

The complete set of statistical conclusions for both location and dispersion comparisons appears in Table 2 arranged by programming aspect. Instances of non-null (or alternative) conclusions--those indicating some distinction among the groups on the basis of a measured programming aspect--are listed by outcome in Tables 4.1 (for location comparisons) and 4.2 (for dispersion comparisons).

Examination of Table 2 immediately demonstrates that a large number of the programming aspects considered in this study, especially product aspects, failed to show any distinction between the groups. This low "yield" is not surprising, especially among product aspects, and may be attributed to the partially exploratory nature of the study, the small sample sizes, and the general coarseness of many of the aspects considered. The issue of these null outcome occurrences and their significance is treated more thoroughly in the next subsection, Impact Evaluation.

It is worth noting, however, that several of the null conclusions may indicate characteristics inherent to the application itself. As one example, the basic symbol-table/scanner/parser/code-generator nature of a compiler strongly influences the way the system is modularized and thus practically determines the number of modules in the final product (give or take some occasional slight variation due to other design decisions).

Impact Evaluation

The collective impact of these statistical conclusions

Table 2. Statistical Conclusions

N.B. A simple pair of equal signs (= =) appears in place of the null outcome AI = AT = DT in order to avoid cluttering the table. The asterisks at the left margin mark the confirmatory aspects; exploratory aspects are unmarked.

	programming aspect	location comparison outcome	location critical level	dispersion comparison outcome	dispersion critical level
	rudimentary process aspects				
*	COMPUTER JOB STEPS	DT < AI = AT	0.003	= =	
*	MODULE COMPILATION	DT < AI = AT	0.022	= =	
*	UNIQUE	DT < AI = AT	0.011	= =	
	IDENTICAL	= =		= =	
*	PROGRAM EXECUTION	DT < AI = AT	0.022	= =	
	MISCELLANEOUS	DT < AI = AT	0.144	AT = DT < AI	0.077
*	ESSENTIAL	DT < AI = AT	0.003	= =	
	AVERAGE UNIQUE COMPILATIONS PER MODULE	DT < AI = AT	0.088	= =	
	MAX. UNIQUE COMPILATIONS F.A.O. MODULE	DT < AI = AT	0.118	DT < AI < AT	0.051
	elaborative process aspects				
*	PROGRAM CHANGES	DT < AI < AT	0.184	= =	
	rudimentary product aspects				
*	MODULES	= =		= =	
*	SEGMENTS	AI < AT = DT	0.063	= =	
	SEGMENT TYPE COUNTS :				
	FUNCTION	= =		= =	
	PROCEDURE	= =		= =	
	SEGMENT TYPE PERCENTAGES :				
	FUNCTION	= =		= =	
	PROCEDURE	= =		= =	
	AVERAGE SEGMENTS PER MODULE	= =		DT = AI < AT	0.021
*	LINES	AI < DT < AT	0.119	= =	

MAX. is an abbreviation for MAXIMUM
F.A.O. is an abbreviation for FOR ANY ONE

TABLE 2

*	STATEMENTS	= =		AT < DT = AI	0.195
	STATEMENT TYPE COUNTS :				
	:=	= =		= =	
*	IF	DT = AI < AT	0.078	= =	
*	CASE	= =		= =	
*	WHILE	= =		= =	
*	EXIT	= =		= =	
	(PROC)CALL	= =		DT < AI = AT	0.032
	NONINTRINSIC	= =		DT < AI = AT	0.186
	INTRINSIC	DT = AI < AT	0.173	= =	
*	RETURN	DT = AI < AT	0.086	DT < AI < AT	0.139
	STATEMENT TYPE PERCENTAGES :				
	:=	= =		= =	
*	IF	DT = AI < AT	0.106	= =	
*	CASE	= =		= =	
*	WHILE	= =		= =	
*	EXIT	= =		= =	
	(PROC)CALL	= =		= =	
	NONINTRINSIC	= =		= =	
	INTRINSIC	= =		= =	
*	RETURN	= =		DT = AI < AT	0.040
*	AVERAGE STATEMENTS PER SEGMENT	AT = DT < AI	0.170	= =	
*	AVERAGE STATEMENT NESTING LEVEL	= =		= =	
*	DECISIONS	DT = AI < AT	0.146	= =	
	FUNCTION CALLS	= =		= =	
	NONINTRINSIC	= =		= =	
	INTRINSIC	= =		= =	
*	TOKENS	= =		= =	
*	AVERAGE TOKENS PER STATEMENT	= =		AI = AT < DT	0.106
	INVOCATIONS	= =		AT = DT < AI	0.020
	FUNCTION	= =		= =	
	NONINTRINSIC	= =		= =	
	INTRINSIC	= =		= =	
	PROCEDURE	= =		DT < AI = AT	0.032
	NONINTRINSIC	= =		DT < AI = AT	0.186
	INTRINSIC	DT = AI < AT	0.173	= =	
	NONINTRINSIC	= =		AT = DT < AI	0.051
	INTRINSIC	DT = AI < AT	0.043	= =	
	AVG. INVOCATIONS PER (CALLING) SEGMENT	= =		= =	
	FUNCTION	= =		= =	
	NONINTRINSIC	= =		= =	
	INTRINSIC	= =		= =	
	PROCEDURE	= =		= =	
	NONINTRINSIC	= =		= =	
	INTRINSIC	= =		DT < AI = AT	0.065
	NONINTRINSIC	AT = DT < AI	0.169	= =	
	INTRINSIC	= =		= =	

TABLE 2

TABLE 2

AVG. INVOCATIONS PER (CALLED) SEGMENT	AT = DT < AI	0.169			
FUNCTION	AT = DT < AI	0.193	AT < DT = AI	0.141	
PROCEDURE					
DATA VARIABLES	AI < AT = DT	0.069			
DATA VARIABLE SCOPE COUNTS :					
* GLOBAL	AI < AT = DT	0.147	AI = AT < DT	0.124	
ENTRY					
MODIFIED					
UNMODIFIED					
NONENTRY					
MODIFIED					
UNMODIFIED					
MODIFIED	AI < AT = DT	0.161			
UNMODIFIED					
NONGLOBAL					
* PARAMETER	AI < AT = DT	0.127	AI = AT < DT	0.106	
VALUE					
REFERENCE			AI < AT = DT	0.019	
* LOCAL					
DATA VARIABLE SCOPE PERCENTAGES :					
* GLOBAL			AI = AT < DT	0.075	
ENTRY					
MODIFIED					
UNMODIFIED					
NONENTRY					
MODIFIED			DT < AI = AT	0.021	
UNMODIFIED					
MODIFIED					
UNMODIFIED					
NONGLOBAL			AI = AT < DT	0.075	
* PARAMETER	AI < AT = DT	0.150	AI = AT < DT	0.055	
VALUE			AI = AT < DT	0.094	
REFERENCE			AI < AT = DT	0.152	
* LOCAL	AT = DT < AI	0.109			
AVERAGE GLOBAL VARIABLES PER MODULE					
ENTRY					
NONENTRY					
MODIFIED			DT = AI < AT	0.110	
UNMODIFIED					
AVERAGE NONGLOBAL VARIABLES PER SEGMENT					
PARAMETER	AI < AT = DT	0.174			
LOCAL					
PARAMETER PASSAGE TYPE PERCENTAGES :					
VALUE			AI < AT = DT	0.160	
REFERENCE			AI < AT = DT	0.160	
(SEGMENT,GLOBAL) ACTUAL USAGE PAIRS					
ENTRY					
MODIFIED					
UNMODIFIED					
NONENTRY					
MODIFIED					

TABLE 2

TABLE 2

UNMODIFIED		AT < DT = AI	0.106
MODIFIED			
UNMODIFIED			
(SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS	AI < AT = DT : 0.122	AI < DT < AT	0.052
ENTRY			
MODIFIED			
UNMODIFIED		AI < AT = DT	0.078
NONENTRY		DT = AI < AT	0.051
MODIFIED		AI < DT < AT	0.172
UNMODIFIED			
MODIFIED			
UNMODIFIED			
* (SEGMENT,GLOBAL) USAGE PAIR REL.PERCENT.			
ENTRY	AT < DT < AI : 0.117		
MODIFIED	AT < DT < AI : 0.125		
UNMODIFIED			
NONENTRY			
MODIFIED			
UNMODIFIED	AT < DT = AI : 0.154		
MODIFIED			
UNMODIFIED			

AVG. is an abbreviation for AVERAGE
REL.PERCENT. is an abbreviation for RELATIVE PERCENTAGE

elaborative product aspects

CYCLOMATIC COMPLEXITY :			
SIMPPRED-NCASE VARIATION :			
TOTAL			
#SEGS:CC>=10		AI < AT = DT	0.116
0.5 Q ANTILE POINT VALUE	AT = DT < AI : 0.185		
0.5 Q ANTILE TAIL AVERAGE			
0.7 Q ANTILE POINT VALUE	AT = DT < AI : 0.072		
0.7 Q ANTILE TAIL AVERAGE	AT = DT < AI : 0.180		
0.8 Q ANTILE POINT VALUE			
0.8 Q ANTILE TAIL AVERAGE			
0.9 Q ANTILE POINT VALUE			
0.9 Q ANTILE TAIL AVERAGE			
SIMPPRED-LOGCASE VARIATION :			
TOTAL			
#SEGS:CC>=10		DT = AI < AT	0.142
0.5 Q ANTILE POINT VALUE			
0.5 Q ANTILE TAIL AVERAGE	DT < AI = AT : 0.185		
0.7 Q ANTILE POINT VALUE			
0.7 Q ANTILE TAIL AVERAGE	DT < AI = AT : 0.188		
0.8 Q ANTILE POINT VALUE	DT < AI = AT : 0.114		
0.8 Q ANTILE TAIL AVERAGE			
0.9 Q ANTILE POINT VALUE	AT = DT < AI : 0.103		
0.9 Q ANTILE TAIL AVERAGE			

TABLE 2

TABLE 2

COMPPRED-NCASE VARIATION :					
TOTAL	DT = AI < AT	0.105			
#SEGS:CC>=10	= = =		DT = AI < AT	0.129	
0.5 Q ANTILE POINT VALUE	= = =		= = =		
0.5 Q ANTILE TAIL AVERAGE	AT = DT < AI	0.155	= = =		
0.7 Q ANTILE POINT VALUE	DT < AI = AT	0.140	= = =		
0.7 Q ANTILE TAIL AVERAGE	AT = DT < AI	0.154	= = =		
0.8 Q ANTILE POINT VALUE	= = =		= = =		
0.8 Q ANTILE TAIL AVERAGE	AT = DT < AI	0.154	= = =		
0.9 Q ANTILE POINT VALUE	AT = DT < AI	0.134	AT = DT < AI	0.066	
0.9 Q ANTILE TAIL AVERAGE	AT = DT < AI	0.136	= = =		
COMPPRED-LOGCASE VARIATION :					
TOTAL	DT = AI < AT	0.149	= = =		
#SEGS:CC>=10	= = =		DT = AI < AT	0.196	
0.5 Q ANTILE POINT VALUE	= = =		AI < AT = DT	0.151	
0.5 Q ANTILE TAIL AVERAGE	DT < AI = AT	0.197	= = =		
0.7 Q ANTILE POINT VALUE	DT < AI = AT	0.082	= = =		
0.7 Q ANTILE TAIL AVERAGE	DT < AI = AT	0.197	= = =		
0.8 Q ANTILE POINT VALUE	= = =		= = =		
0.8 Q ANTILE TAIL AVERAGE	= = =		= = =		
0.9 Q ANTILE POINT VALUE	AT = DT < AI	0.089	= = =		
0.9 Q ANTILE TAIL AVERAGE	= = =		= = =		
(SEGMENT,GLOBAL,SEGMENT) DATA BINDINGS :					
ACTUAL	= = =		= = =		
SUBFUNCTIONAL	= = =		= = =		
INDEPENDENT	= = =		AI < AT = DT	0.196	
POSSIBLE	DT = AI < AT	0.186	DT = AI < AT	0.152	
RELATIVE PERCENTAGE	= = =		= = =		
SOFTWARE SCIENCE QUANTITIES :					
VOCABULARY	AI < AT = DT	0.089	= = =		
LENGTH	= = =		= = =		
ESTIMATED LENGTH	AI < AT = DT	0.140	= = =		
%DIFFERENCE(N,N^)	= = =		AI = AT < DT	0.156	
VOLUME	= = =		= = =		
INTELLIGENCE CONTENT	= = =		= = =		
ESTIMATED BUGS	= = =		= = =		
1ST CALCULATION METHOD :					
PROGRAM LEVEL	= =		AI = AT < DT	0.027	
DIFFICULTY	= =		= =		
POTENTIAL VOLUME	= =		= =		
LANGUAGE LEVEL	= =		AI = AT < DT	0.148	
EFFORT	= =		AI = AT < DT	0.058	
ESTIMATED TIME	= =		AI = AT < DT	0.058	
2ND CALCULATION METHOD :					
PROGRAM LEVEL	= =		= =		
DIFFICULTY	= =		= =		
POTENTIAL VOLUME	= =		= =		
%DIFFERENCE(V*,I)	= =		= =		
LANGUAGE LEVEL	= =		= =		
EFFORT	= =		= =		
ESTIMATED TIME	= =		= =		

TABLE 2

may be objectively evaluated according to the following statistical principle [Tukey 69, pp. 84-85]. Whenever a series of statistical tests (or experiments) are made, all at a fixed level of significance (for example, 0.10), a corresponding percentage (in the example, 10%) of the tests are expected a priori to reject the null hypothesis in the complete absence of any true effect (i.e., due to chance alone). This expected rejection percentage provides a comparative index of the true impact of the test results as a whole (in the example, a 25% actual rejection percentage would indicate that a truely significant effect, other than chance alone, was operative).

The point here may be illustrated in terms of simple coin-tossing experiments. The nature of statistics itself dictates that, out of a series of 100 separate statistical tests of a hypothetically fair coin at the 0.05 significance level, roughly 5 of those tests would nonetheless indicate that the coin was biased; if only 6 out of 100 tests of a real coin indicate bias at the 0.05 level, those six results have very little impact since the coin is behaving rather unbiasedly over the full set of tests.

This same "multiplicity" principle applies to the statistical conclusions of the study, since they represent the outcomes of a series of separate tests and were assumed in the statistical model to be separate experiments. It is appropriate to evaluate the location and dispersion results separately, since they reflect two separate issues (expectency and predictability) of software development behavior. It is also appropriate to evaluate the process and product results separately. Finally, it is only fair to evaluate the confirmatory aspects as a distinct subset of all aspects examined, since they alone had been honestly considered prior to collecting and analyzing the data.

Details of this impact evaluation for the study's objective results, broken down into the appropriate categories identified above, are presented in the following table. (This table is an excerpt from Table 3, which provides an extensive impact evaluation, broken down hierarchically according to all of the various dichotomies identified for the programming aspects.) The evaluation was performed at the $\alpha = 0.20$ significance level used for screening purposes, hence the expected rejection percentage for any category was 20%. For each category of aspects, the table gives the number of (nonredundant) programming aspects, the expected (rounded to whole numbers) and actual numbers of rejections (of the null conclusion in favor of a directional alternative), and the expected and actual rejection percentages. An asterisk marks those categories demonstrating noticable statistical impact (i.e., actual rejection percentage well above expected rejection percentage).

category	#	exp. # rej.	act. # rej.	exp. rej. %	act. rej. %	
location	189	38	53	20.0	28.0	
process	10	2	9	20.0	90.0	*
confirmatory only	6	1	6	20.0	100.0	*
product	179	36	44	20.0	24.6	
confirmatory only	29	6	12	20.0	41.4	*
confirmatory only	35	7	18	20.0	51.4	*
dispersion	189	38	43	20.0	22.8	
process	10	2	2	20.0	20.0	
confirmatory only	6	1	0	20.0	0.0	
product	179	36	41	20.0	22.9	
confirmatory only	29	6	9	20.0	31.0	*
confirmatory only	35	7	9	20.0	25.7	

\# : number of aspects
exp. # rej. : expected number of rejections
act. # rej. : actual number of rejections
exp. rej. % : expected rejection percentage
act. rej. % : actual rejection percentage

The table shows that the location results, dealing with the expectency of software development behavior, do have statistical impact in several subcategories. Process aspects have more impact than product aspects on the whole,

TABLE 3

Table 3. Statistical Impact Evaluation

category	number of aspects	expect. num. of reject.	actual num. of reject.	expect. reject. percent	actual reject. percent	
location	189	38	53	20.0	26.0	
process	10	2	9	20.0	90.0	*
rudimentary	9	2	8	20.0	88.9	*
confirmatory	5	1	5	20.0	100.0	*
exploratory	4	1	3	20.0	75.0	*
elaborative	1	0	1	20.0	100.0	*
confirmatory	1	0	1	20.0	100.0	*
exploratory	0	0	0	20.0	0.0	
confirmatory	6	1	6	20.0	100.0	*
exploratory	4	1	3	20.0	75.0	*
product	179	36	44	20.0	24.6	
rudimentary	115	23	22	20.0	19.1	
confirmatory	26	5	11	20.0	42.3	*
exploratory	89	18	11	20.0	12.4	
elaborative	64	13	22	20.0	34.4	*
confirmatory	3	1	1	20.0	33.3	*
exploratory	61	12	21	20.0	34.5	*
confirmatory	29	6	12	20.0	41.4	*
exploratory	150	30	32	20.0	21.3	
rudimentary	124	25	30	20.0	24.2	
confirmatory	31	6	16	20.0	51.6	*
exploratory	93	19	14	20.0	15.1	
elaborative	65	13	23	20.0	35.4	*
confirmatory	4	1	2	20.0	50.0	*
exploratory	61	12	21	20.0	34.5	*
confirmatory	35	7	18	20.0	51.4	*
exploratory	154	31	35	20.0	22.7	
dispersion	189	38	43	20.0	22.8	
process	10	2	2	20.0	20.0	
rudimentary	9	2	2	20.0	22.2	
confirmatory	5	1	0	20.0	0.0	
exploratory	4	1	2	20.0	50.0	*
elaborative	1	0	0	20.0	0.0	
confirmatory	1	0	0	20.0	0.0	
exploratory	0	0	0	20.0	0.0	
confirmatory	6	1	0	20.0	0.0	
exploratory	4	1	2	20.0	50.0	*
product	179	36	41	20.0	22.9	
rudimentary	115	23	28	20.0	24.3	
confirmatory	26	5	8	20.0	30.8	*
exploratory	89	18	20	20.0	22.5	
elaborative	64	13	13	20.0	20.3	
confirmatory	3	1	1	20.0	33.3	*
exploratory	61	12	12	20.0	19.7	
confirmatory	29	6	9	20.0	31.0	*
exploratory	150	30	32	20.0	21.3	
rudimentary	124	25	30	20.0	24.2	
confirmatory	31	6	8	20.0	25.8	
exploratory	93	19	22	20.0	23.7	
elaborative	65	13	13	20.0	20.0	
confirmatory	4	1	1	20.0	25.0	
exploratory	61	12	12	20.0	19.7	
confirmatory	35	7	9	20.0	25.7	
exploratory	154	31	34	20.0	22.1	

but when tempered by consideration of the distinction between confirmatory and exploratory aspects, the study's location results bear strong statistical impact for both process and product. They are better explained as the consequence of some true effect related to the experimental treatments, rather than as a random phenomenon.

It is also clear from the table that the dispersion results, dealing with the predictability of software development behavior, have little statistical impact in general. This is due primarily to the diminished power of statistical procedures used to test for dispersion differences, compounded by the small sample sizes involved and the coarseness of many of the programming aspects themselves. The lack of strong statistical impact in this area of the study does not mean that the dispersion issue is unimportant or undeserving of research attention, but rather that it is "a tougher nut to crack" than the location issue. The study's dispersion results are still worth pursuing, however, as possible hints of where differences might exist, provided this disclaimer regarding their impact is heeded.

A Relaxed Differentiation View

As described in Chapter VI, the research framework of possible three-way comparison outcomes provided the basis for converting the statistical results into the statistical conclusions. This framework has two inherent structural characteristics that may be exploited to make additional observations regarding the statistical conclusions. These structural characteristics and the supplemental views of the conclusions that they afford are described here and in the next subsection.

The first structural characteristic is that each

Table 4.1 Non-Null Conclusions, for Location Comparisons, arranged by outcome

comparison outcome	critical level	#	programming aspect
AI < AT = DT		10	
	0.063		SEGMENTS
	0.069		DATA VARIABLES
	0.147		DATA VARIABLE SCOPE COUNTS \ GLOBAL
	0.161		DATA VARIABLE SCOPE COUNTS \ GLOBAL \ MODIFIED
	0.127		DATA VARIABLE SCOPE COUNTS \ NONGLOBAL \ PARAMETER
	0.150		DATA VARIABLE SCOPE PERCENTAGES \ NONGLOBAL \ PARAMETER
	0.174		AVERAGE NONGLOBAL VARIABLES PER SEGMENT \ PARAMETER
	0.122		(SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS
	0.089		SOFTWARE SCIENCE QUANTITIES \ VOCABULARY
	0.140		SOFTWARE SCIENCE QUANTITIES \ ESTIMATED LENGTH
AT = DT < AI		15	
	0.170		AVERAGE STATEMENTS PER SEGMENT
	0.169		AVERAGE INVOCATIONS PER (CALLING) SEGMENT \ NONINTRINSIC
	0.169		AVERAGE INVOCATIONS PER (CALLED) SEGMENT
	0.193		AVERAGE INVOCATIONS PER (CALLED) SEGMENT \ FUNCTION
	0.109		DATA VARIABLE SCOPE PERCENTAGES \ NONGLOBAL \ LOCAL
	0.185		CYCLOMATIC COMPLEXITY \ SIMPPRED-NCASE VARIATION \ 0.5 QUANTILE POINT VALUE
	0.072		CYCLOMATIC COMPLEXITY \ SIMPPRED-NCASE VARIATION \ 0.7 QUANTILE POINT VALUE
	0.180		CYCLOMATIC COMPLEXITY \ SIMPPRED-NCASE VARIATION \ 0.7 QUANTILE TAIL AVERAGE
	0.103		CYCLOMATIC COMPLEXITY \ SIMPPRED-LOGCASE VARIATION \ 0.9 QUANTILE POINT VALUE
	0.155		CYCLOMATIC COMPLEXITY \ COMPPRED-NCASE VARIATION \ 0.5 QUANTILE TAIL AVERAGE
	0.154		CYCLOMATIC COMPLEXITY \ COMPPRED-NCASE VARIATION \ 0.7 QUANTILE TAIL AVERAGE
	0.154		CYCLOMATIC COMPLEXITY \ COMPPRED-NCASE VARIATION \ 0.8 QUANTILE TAIL AVERAGE
	0.134		CYCLOMATIC COMPLEXITY \ COMPPRED-NCASE VARIATION \ 0.9 QUANTILE POINT VALUE
	0.136		CYCLOMATIC COMPLEXITY \ COMPPRED-NCASE VARIATION \ 0.9 QUANTILE TAIL AVERAGE
	0.089		CYCLOMATIC COMPLEXITY \ COMPPRED-LOGCASE VARIATION \ 0.9 QUANTILE POINT VALUE
AI < DT = AI		1	
	0.154		(SEGMENT,GLOBAL) USAGE PAIRS RELATIVE PERCENTAGE \ NONENTRY \ UNMODIFIED
DT = AI < AT		10	
	0.078		STATEMENT TYPE COUNTS \ IF
	0.173		STATEMENT TYPE COUNTS \ (PROC)CALL \ INTRINSIC
	0.086		STATEMENT TYPE COUNTS \ RETURN
	0.106		STATEMENT TYPE PERCENTAGES \ IF
	0.146		DECISIONS
	0.175		INVOCATIONS \ PROCEDURE \ INTRINSIC
	0.043		INVOCATIONS \ INTRINSIC
	0.105		CYCLOMATIC COMPLEXITY \ COMPPRED-NCASE VARIATION \ TOTAL
	0.149		CYCLOMATIC COMPLEXITY \ COMPPRED-LOGCASE VARIATION \ TOTAL
	0.186		(SEGMENT,GLOBAL,SEGMENT) DATA BINDINGS \ POSSIBLE
DT < AI = AT		15	
	0.003		COMPUTER JOB STEPS
	0.022		COMPUTER JOB STEPS \ MODULE COMPILATION
	0.011		COMPUTER JOB STEPS \ MODULE COMPILATION \ UNIQUE
	0.022		COMPUTER JOB STEPS \ PROGRAM EXECUTION
	0.144		COMPUTER JOB STEPS \ MISCELLANEOUS
	0.003		COMPUTER JOB STEPS \ ESSENTIAL
	0.088		AVERAGE UNIQUE COMPILATIONS PER MODULE
	0.116		MAXIMUM UNIQUE COMPILATIONS FOR ANY ONE MODULE
	0.165		CYCLOMATIC COMPLEXITY \ SIMPPRED-LOGCASE VARIATION \ 0.5 QUANTILE TAIL AVERAGE
	0.183		CYCLOMATIC COMPLEXITY \ SIMPPRED-LOGCASE VARIATION \ 0.7 QUANTILE TAIL AVERAGE
	0.114		CYCLOMATIC COMPLEXITY \ SIMPPRED-LOGCASE VARIATION \ 0.8 QUANTILE POINT VALUE
	0.140		CYCLOMATIC COMPLEXITY \ COMPPRED-NCASE VARIATION \ 0.7 QUANTILE POINT VALUE
	0.197		CYCLOMATIC COMPLEXITY \ COMPPRED-LOGCASE VARIATION \ 0.5 QUANTILE TAIL AVERAGE
	0.062		CYCLOMATIC COMPLEXITY \ COMPPRED-LOGCASE VARIATION \ 0.7 QUANTILE POINT VALUE
	0.197		CYCLOMATIC COMPLEXITY \ COMPPRED-LOGCASE VARIATION \ 0.7 QUANTILE TAIL AVERAGE
AI = AT < DT		0	
AI < AT < DT		0	
AI < DT < AT		1	
	0.119		LINES
AT < DT < AI		2	
	0.117		(SEGMENT,GLOBAL) USAGE PAIR RELATIVE PERCENTAGE \ ENTRY
	0.123		(SEGMENT,GLOBAL) USAGE PAIR RELATIVE PERCENTAGE \ ENTRY \ MODIFIED
AT < AI < DT		0	
DT < AI < AT		1	
	0.184		PROGRAM CHANGES
DT < AT < AI		0	

this column records the frequency of occurrence for each comparison outcome

TABLE 4

Table 4.2 Non-Null Conclusions, for Dispersion Comparisons, arranged by outcome

comparison outcome	critical level	#	programming aspect
AI < AT = DT		6	
	0.019		DATA VARIABLE SCOPE COUNTS \ NONGLOBAL \ PARAMETER \ REFERENCE
	0.152		DATA VARIABLE SCOPE PERCENTAGES \ NONGLOBAL \ PARAMETER \ REFERENCE
	0.160		PARAMETER PASSAGE TYPE PERCENTAGES \ VALUE
	0.160		PARAMETER PASSAGE TYPE PERCENTAGES \ REFERENCE
	0.078		(SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS \ NONENTRY
	0.116		CYCLOMATIC COMPLEXITY \ SIMPPRED-NCASE VARIATION \ #SEGS:CC>=10
	0.151		CYCLOMATIC COMPLEXITY \ COMPPRED-LOGCASE VARIATION \ 0.5 QUANTILE POINT VALUE
	0.196		(SEGMENT,GLOBAL,SEGMENT) DATA BINDINGS \ ACTUAL \ INDEPENDENT
AT = DT < AI		4	
	0.077		COMPUTER JOB STEPS \ MISCELLANEOUS
	0.020		INVOCATIONS
	0.051		INVOCATIONS \ NONINTRINSIC
	0.066		CYCLOMATIC COMPLEXITY \ COMPPRED-NCASE VARIATION \ 0.9 QUANTILE POINT VALUE
AT < DT = AI		3	
	0.195		STATEMENTS
	0.141		AVERAGE INVOCATIONS PER (CALLED) SEGMENT \ FUNCTION
	0.106		(SEGMENT,GLOBAL) ACTUAL USAGE PAIRS \ MODIFIED
DT = AI < AT		8	
	0.021		AVERAGE SEGMENTS PER MODULE
	0.040		STATEMENT TYPE PERCENTAGES \ RETURN
	0.110		AVERAGE GLOBAL VARIABLES PER MODULE \ MODIFIED
	0.051		(SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS \ NONENTRY \ MODIFIED
	0.142		CYCLOMATIC COMPLEXITY \ SIMPPRED-LOGCASE VARIATION \ #SEGS:CC>=10
	0.129		CYCLOMATIC COMPLEXITY \ COMPPRED-NCASE VARIATION \ #SEGS:CC>=10
	0.196		CYCLOMATIC COMPLEXITY \ COMPPRED-LOGCASE VARIATION \ #SEGS:CC>=10
	0.152		(SEGMENT,GLOBAL,SEGMENT) DATA BINDINGS \ POSSIBLE
DT < AI = AT		6	
	0.032		STATEMENT TYPE COUNTS \ (PROC)CALL
	0.186		STATEMENT TYPE COUNTS \ (PROC)CALL \ NONINTRINSIC
	0.032		INVOCATIONS \ PROCEDURE
	0.186		INVOCATIONS \ PROCEDURE \ NONINTRINSIC
	0.065		AVERAGE INVOCATIONS PER (CALLING) SEGMENT \ PROCEDURE \ INTRINSIC
	0.021		DATA VARIABLE SCOPE PERCENTAGES \ GLOBAL \ NONENTRY \ MODIFIED
AI = AT < DT		12	
	0.106		AVERAGE TOKENS PER STATEMENT
	0.124		DATA VARIABLE SCOPE COUNTS \ GLOBAL
	0.106		DATA VARIABLE SCOPE COUNTS \ NONGLOBAL \ PARAMETER
	0.075		DATA VARIABLE SCOPE PERCENTAGES \ GLOBAL
	0.075		DATA VARIABLE SCOPE PERCENTAGES \ NONGLOBAL
	0.055		DATA VARIABLE SCOPE PERCENTAGES \ NONGLOBAL \ PARAMETER
	0.094		DATA VARIABLE SCOPE PERCENTAGES \ NONGLOBAL \ PARAMETER \ VALUE
	0.156		SOFTWARE SCIENCE QUANTITIES \ 2DIFFERENCE(N,N)
	0.027		SOFTWARE SCIENCE QUANTITIES \ 1ST CALCULATION METHOD \ PROGRAM LEVEL
	0.148		SOFTWARE SCIENCE QUANTITIES \ 1ST CALCULATION METHOD \ LANGUAGE LEVEL
	0.058		SOFTWARE SCIENCE QUANTITIES \ 1ST CALCULATION METHOD \ EFFORT
	0.059		SOFTWARE SCIENCE QUANTITIES \ 1ST CALCULATION METHOD \ ESTIMATED TIME
AI < AT < DT		0	
AT < DT < AI		2	
	0.052		(SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS
	0.172		(SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS \ NONENTRY \ UNMODIFIED
AT < DT < AI		0	
AT < AI < DT		0	
DT < AI < AT		2	
	0.051		MAXIMUM UNIQUE COMPILATIONS OR ANY ONE MODULE
	0.139		STATEMENT TYPE COUNTS \ RETURN
DT < AT < AI		0	

this column records the frequency of occurre - for each comparison outcome

TABLE 4

completely differentiated outcome is related to a specific pair of partially differentiated outcomes, as shown in the lattice of Figure 7.1. For example, AI < AT < DT, a completely differentiated outcome, naturally weakens to either AI < AT = DT or AI = AT < DT, two partially differentiated outcomes.

Each completely differentiated outcome consists of three pairwise differences (AI < AT, AT < DT, AI < DT in the example), while each partially differentiated outcome consists of only two pairwise differences plus one pairwise equality (AI < DT, AI < AT, AT = DT and AI < DT, AT < DT, AI = AT in the example). The "outer" difference of the completely differentiated outcome (AI < DT in the example) is common to both partially differentiated outcomes, while each partially differentiated outcome focuses attention on one of the two "inner" differences (AI < AT and AT < DT in the example) to the exclusion of the other "inner" difference which is "relaxed" to an equality. Within a statistical environment or model which places a premium on claiming differences instead of equalities, a partially differentiated outcome is a safer statement, containing less error-prone information than a completely differentiated outcome. Since these outcomes represent statistical conclusions, the same data scores which support a completely differentiated outcome at a certain critical level also support each of the two related partially differentiated outcomes at lower critical levels.

Thus, every completely differentiated conclusion may also be considered as two (more significant) partially differentiated conclusions, each of these three conclusions having equal and complete statistical legitimacy. The "outer" difference of a completely differentiated conclusion is, of course, stronger than either of its two "inner"

differences; but the strengths of the two "inner" differences (relative to each other) will vary in accordance with the data scores and indeed are reflected in the significance levels of the two corresponding partially differentiated conclusions (relative to each other). Tables 5.1 and 5.2 give the details of this "relaxed differentiation" analysis for each of the completely differentiated conclusions found in the study, and an English paraphrase appears in the two paragraphs immediately below. All of the partially differentiated conclusions listed in these tables should be added to those presented in Tables 2 and 4; they deserve full consideration in any analysis or interpretation of the study's findings. However, in the case that one of a partially differentiated pair is noticeably stronger than the other, it is fair to consider only the stronger one for the purpose of analysis or interpretation dealing primarily with partially differentiated outcomes, since the study is mainly concerned with the most pronounced difference afforded by each aspect's data scores.

On location comparisons, four programming aspects yielded completely differentiated conclusions. They are "relaxed" to partially differentiated conclusions as follows:

1. From DT < AI < AT on the PROGRAM CHANGES aspect, the DT < AI = AT conclusion dwarfs the DT = AI < AT conclusion with respect to level of significance.
2. The DT < AT difference is more pronounced than the AI < DT difference from AI < DT < AT on the LINES aspect.
3. AT < DT < AI on the (SEGMENT,GLOBAL) USAGE PAIR RELATIVE PERCENTAGE\ENTRY aspect is more appropriately "relaxed" to the AT < DT = AI conclusion than to the AT = DT < AI conclusion.

TABLE 5

Table 5.1 Relaxed Differentiation for Location Comparisons

programming aspect	completely differentiated conclusion		partially differentiated conclusions	
	comparison outcome	critical level	comparison outcome	critical level
PROGRAM CHANGES	DT < AI < AT	0.184	DT < AI = AT DT = AI < AT	0.003 0.184
LINES	AI < DT < AT	0.119	DT = AI < AT AI < AT = DT	0.061 0.113
(SEGMENT,GLOBAL) USAGE PAIR RELATIVE PERCENTAGE \ ENTRY	AT < DT < AI	0.117	AT < DT = AI AT = DT < AI	0.082 0.111
(SEGMENT,GLOBAL) USAGE PAIR RELATIVE PERCENTAGE \ ENTRY \ MODIFIED	AT < DT < AI	0.123	AT < DT = AI AT = DT < AI	0.113 0.113

Table 5.2 Relaxed Differentiation for Dispersion Comparisons

programming aspect	completely differentiated conclusion		partially differentiated conclusions	
	comparison outcome	critical level	comparison outcome	critical level
MAXIMUM UNIQUE COMPILATIONS FOR ANY ONE MODULE	DT < AI < AT	0.051	DT < AI = AT DT = AI < AT	0.003 0.051
STATEMENT TYPE COUNTS \ RETURN	DT < AI < AT	0.139	DT = AI < AT DT < AI = AT	0.003 0.139
(SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS	AI < DT < AT	0.052	AI < AT = DT DT = AI < AT	0.020 0.051
(SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS \ NONENTRY \ UNMODIFIED	AI < DT < AT	0.172	AI < AT = DT DT = AI < AT	0.116 0.156

TABLE 5

4. The AT < DT and DT < AI differences from AT < DT < AI on the (SEGMENT,GLOBAL) USAGE PAIR RELATIVE PERCENTAGE\ ENTRY\MODIFIED aspect are equally strong.

On dispersion comparisons, four programming aspects yielded completely differentiated conclusions. They are "relaxed" to partially differentiated conclusions as follows:

1. The DT < AI difference is much more pronounced than the AI < AT difference from DT < AI < AT on the MAXIMUM UNIQUE COMPILATIONS FOR ANY ONE MODULE aspect.
2. From DT < AI < AT on the STATEMENT TYPE COUNTS\RETURN aspect, the DT = AI < AT conclusion dwarfs the DT < AI = AT conclusion with respect to level of significance.
3. AI < DT < AT on the (SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS aspect is more appropriately "relaxed" to the AI < AT = DT conclusion than to the DT = AI < AT conclusion.
4. The AI < DT difference is more pronounced than the DT < AT difference from AI < DT < AT on the (SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS\NONENTRY\ UNMODIFIED aspect.

A Directionless View

The second structural characteristic of the possible outcome framework is that the outcomes may be classified into another closely related set of directionless outcomes, as shown in the lattice of Figure 3.2. For example, AI < AT = DT and AT = DT < AI, two directional partially differentiated outcomes, both correspond to AI ≠ AT = DT, a nondirectional partially differentiated outcome. All six of the directional completely differentiated outcomes correspond to the single nondirectional completely

differentiated outcome AI ≠ AT ≠ DT.

By emphasizing just the existence and not the direction of distinctions between the treatment groups, these directionless outcome categories focus attention on the original research issue of discovering which observable programming aspects differentiate among the three programming environments. In particular, there are three nondirectional partially differentiated outcomes (each of the form "one group different from the other two which are similar"), and it is noteworthy to observe just what set of programming aspects supports each of these basic distinctions. (Table 4 is arranged so that the directional distinctions listed there can be readily coalesced by eye into directionless categories.) It is revealing to note that, with one exception, the directionless distinctions on location comparisons segregate cleanly along the process-versus-product dichotomy line: all of the product distinctions fall into the AI ≠ AT = DT or AT ≠ DT = AI categories, while the process distinctions consistently fall into the DT ≠ AI = AT category. Interestingly enough, the one exception is that a number of the cyclomatic complexity metric variations (which are product aspects) show the DT ≠ AI = AT directionless outcome (which otherwise characterizes only process aspect distinctions).

Individual Highlights

The purpose of this concluding section is to point out what seem to be the "top ten" (well, eleven and nine) most noteworthy conclusions from among the study's objective results. These conclusions are interesting individually, either because the programming aspect merits attention or because the difference in its expectency or predictability is pronounced (as indicated by a low critical significance

level) in the experimental sample data.

Noteworthy location distinctions are mentioned below.

1. According to the DT < AI = AT outcome on the COMPUTER JOB STEPS aspect, the disciplined teams used very noticeably fewer computer job steps (i.e., module compilations, program executions, and miscellaneous job steps) than either the ad hoc individuals or the ad hoc teams.
2. This same difference was apparent in the total number of module compilations, the number of unique (i.e., not identical to a previous compilation) module compilations, the number of program executions, and the number of essential job steps (i.e., unique module compilations plus program executions), according to the DT < AI = AT outcomes on the COMPUTER JOB STEPS\MODULE COMPILATION, COMPUTER JOB STEPS\MODULE COMPILATION\ UNIQUE, COMPUTER JOB STEPS\PROGRAM EXECUTION, and COMPUTER JOB STEPS\ESSENTIAL aspects, respectively.
3. According to the DT < AI = AT outcome on the PROGRAM CHANGES aspect, the disciplined teams required fewer textual revisions to build and debug the software than the ad hoc individuals and the ad hoc teams.
4. There was a definite trend for the ad hoc individuals to have produced fewer total symbolic lines (including comments, compiler directives, statements, declarations, etc.) than the disciplined teams who produced fewer than the ad hoc teams, according to the AI < DT < AT outcome on the LINES aspect.
5. According to the AI < AT = DT outcome on the SEGMENTS aspect, the ad hoc individuals organized their software into noticeably fewer routines (i.e., functions or procedures) than either the ad hoc teams or the disciplined teams.
6. The ad hoc individuals displayed a trend toward having a

greater number of executable statements per routine than did either the ad hoc teams or the disciplined teams, according to the AT = DT < AI outcome on the AVERAGE STATEMENTS PER SEGMENT aspect.

7. According to the DT = AI < AT outcomes on the STATEMENT TYPE COUNTS\IF and STATEMENT TYPE PERCENTAGE\IF aspects, both the ad hoc individuals and the disciplined teams coded noticeably fewer IF statements than the ad hoc teams, in terms of both total number and percentage of total statements.

8. According to the DT = AI < AT outcome on the DECISIONS aspect, both the ad hoc individuals and the disciplined teams tended to code fewer decisions (i.e., IF, WHILE, or CASE statements) than the ad hoc teams.

9. Both the ad hoc teams and the disciplined teams declared a noticeably larger number of data variables (i.e., scalars or arrays of scalars) than the ad hoc individuals, according to the AI < AT = DT outcome on the DATA VARIABLES aspect.

10. According to the AT = DT < AI outcome on the DATA VARIABLE SCOPE PERCENTAGES\NONGLOBAL\LOCAL aspect, the ad hoc individuals had a larger percentage of local variables compared to the total number of declared data variables than either the ad hoc teams or the disciplined teams.

11. There was a slight trend for both the ad hoc individuals and the disciplined teams to have fewer potential data bindings (i.e., possible communication paths between segments via global variables, as allowed by the software's modularization) than the ad hoc teams, according to the DT = AI < AT outcome on the (SEGMENT,GLOBAL,SEGMENT) DATA BINDINGS\POSSIBLE aspect.

Noteworthy <u>dispersion</u> distinctions are mentioned below.

1. There was a noticeable difference in variability, with

the disciplined teams less than the ad hoc individuals less than the ad hoc teams, in the maximum number of unique compilations for any one module, according to the DT < AI < AT outcome on the MAXIMUM UNIQUE COMPILATIONS FOR ANY ONE MODULE aspect.

2. The ad hoc individuals exhibited noticeably greater variation than either the ad hoc teams or the disciplined teams in the number of miscellaneous job steps (i.e., auxiliary compilations or executions of something other than the final software project), according to the AT = DT < AI outcome on the COMPUTER JOB STEPS\MISCELLANEOUS aspect.
3. According to the DT = AI < AT outcome on the AVERAGE SEGMENTS PER MODULE aspect, the ad hoc individuals and the disciplined teams both exhibited noticeably less variation in the average number of routines per module than the ad hoc teams.
4. According to the DT = AI < AT outcomes on the STATEMENT TYPE COUNTS\RETURN and STATEMENT TYPE PERCENTAGES\RETURN aspects, the ad hoc teams showed rather noticeably greater variability in the number (both raw count and normalized percentage) of RETURN statements coded than both the disciplined teams and the ad hoc individuals.
5. In the number of calls to programmer-defined routines, the ad hoc individuals displayed noticeably greater variation than both the ad hoc teams and the disciplined teams, according to the AT = DT < AI outcome on the INVOCATIONS\NONINTRINSIC aspect.
6. According to the DT < AI = AT outcome on the DATA VARIABLES SCOPE PERCENTAGES\GLOBAL\NONENTRY\MODIFIED aspect, the disciplined teams displayed noticeably smaller variation than either the ad hoc individuals or the ad hoc teams in the percentage of commonplace (i.e., ordinary scope and modified during execution)

global variables compared to the total number of data variables declared.

7. The ad hoc individuals displayed noticeably less variation in the number of formal parameters passed by reference than both the ad hoc teams and the disciplined teams, according to the AI < AT = DT outcome on the DATA VARIABLE SCOPE COUNTS\NONGLOBAL\PARAMETER\REFERENCE aspect.

8. According to the AI < DT < AT outcome on the (SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS aspect, there was a noticeable difference in variability, with the ad hoc individuals less than the disciplined teams less than the ad hoc teams, for the total number of possible segment-global usage pairs (i.e., occurrences of the situation where a global variable could be modified or accessed by a segment).

9. According to the DT = AI < AT outcome on the (SEGMENT,GLOBAL,SEGMENT) DATA BINDINGS\POSSIBLE aspect, the ad hoc teams tended toward greater variability than either the ad hoc individuals or the disciplined teams in the number of potential data bindings.

CHAPTER VIII

VIII. INTERPRETIVE RESULTS

This chapter reports the interpretive results of the study, namely the research interpretations based on the conclusions presented in Chapter VII. The tone of discussion here is purposely somewhat subjective and opinionated, since the study's most important results are derived from interpreting the experiment's immediate findings in view of the study's overall goals. These interpretations also express the researcher's own estimation of the study's implications and general import according to his professional intuitions about programming and software.

The interpretations presented here are neither exhaustive nor unique. They only touch upon certain overall issues and generally avoid attaching meaning to or giving explanation for individual aspects or outcomes. It is anticipated that the reader and other researchers might formulate additional or alternative interpretations of the study's factual findings, using their own intuitive judgments.

Two distinct sets of research interpretations are discussed in the remainder of this chapter. The first set states general trends in the conclusions according to the basic suppositions of the study. The second set states general trends in the conclusions according to a classification of the programming aspects which reflects certain abstract programming notions (e.g., cost, modularity, data organizations, etc.).

According to Basic Suppositions

The study's "basic suppositions" (or "hypotheses") are

a set of simpleminded a priori expectations regarding differences among the experimental programming environments for location and dispersion comparisons on process and product aspects. These basic suppositions are stated in the following table:

Basic Suppositions	for Location Comparisons	for Dispersion Comparisons
on Process Aspects	DT < AI = AT	DT < AI = AT
on Product Aspects	DT = AI < AT or AT < DT = AI	DT = AI < AT or AT < DT = AI

The basic suppositions are founded upon "general beliefs" regarding software phenomena, which had been formulated by the researcher prior to conducting the experiment. These general beliefs state that

(a) methodological discipline is the key influence on the general efficiency of the process;

(b) the disciplined methodology reduces the cost and complexity of the process and enhances the predictability of the process as well;

(c) the preferred direction for both location and dispersion differences on process aspects is clear and undebatable, because of the familiarity of the process aspects and the direct applicability of expected values and variances in terms of average cost estimates and tightness of cost estimates;

(d) "mental cohesiveness" (or conceptual integrity [Brooks 75, pp. 41-50]) is the key influence on the general quality of the product;

(e) a programming team is naturally burdened (relative to an individual programmer) by the organizational overhead and risk of error-prone misunderstanding inherent in coordinating and interfacing the thoughts and efforts of those on the team;

(f) the disciplined methodology induces an effective mental

cohesiveness, enabling a programming team to behave more like an individual programmer with respect to conceptual control over the program, its design, its structure, etc., because of the discipline's antiregressive, complexity-controlling [Belady and Lehman 76, p. 245] effect that compensates for the inherent organizational overhead of a team; and

(g) the preferred direction for both location and dispersion differences on product aspects is not always clear, because of the unfamiliarity of many of the product aspects and a general lack of understanding regarding the implication of dispersion for product aspects.

In view of the general beliefs and basic suppositions stated above, each possible comparison outcome (cf. Figure 3) may be regarded as "voting" either for or against a given basic supposition (or as "abstaining"), depending on whether that outcome would substantiate or contravene the corresponding general beliefs. For process aspects,

(1) outcome DT < AI = AT obviously affirms the supposition;

(2) outcomes DT < AI < AT or DT < AT < AI, which are completely differentiated variations of the supposition's main theme, indirectly affirm the supposition, especially when DT < AI = AT is the stronger of the corresponding partially differentiated outcome pair;

(3) outcome AI = AT = DT may negative the supposition, or it may be considered an abstention for any one of several reasons

(it is possible that (a) the aspect's critical level is not low enough, so it defaults to the null outcome; (b) the aspect reflects something characteristic of the application/task or another factor common to

all the groups in the experiment; or (c) the aspect measures something fundamental to software development phenomena in general and would always result in the null outcome); and

(4) all other outcomes -- AI < AT < DT, AI < DT < AT, AT < DT < AI, AT < AI < DT, AI ≠ AT = DT (AI < AT = DT, AT = DT < AI), AT ≠ AI = DT (AT < DT = AI, DT = AI < AT), and AI = AT < DT -- negative the suppostion.

For product aspects,

(1) outcomes AT ≠ DT = AI (AT < DT = AI, DT = AI < AT) obviously affirm the suppostion;

(2) outcomes AI < DT < AT or AT < DT < AI, which may be considered approximations to the supposition (DT is distinct from AT but falls short of AI, due to lack of experience or maturity in the disciplined methodology), indirectly affirm the supposition, especially when DT = AI < AT or AT < DT = AI (respectively) is the stronger of the corresponding partially differentiated outcome pair;

(3) outcome AI = AT = DT may negative the supposition, or it may be considered an abstention for any one of several reasons

(it is possible that (a) the aspect's critical level is not low enough, so it defaults to the null outcome; (b) the aspect reflects something characteristic of the application/task or anotner factor common to all the groups in the experiment; (c) the aspect measures something fundamental to software development phenomena in general and would always result in the null outcome; or (d) several of the study's hit-and-miss collection of exploratory product aspects are

duds and may be ignored as useless software measures);

(4) outcomes AI < AT < DT, AT < AI < DT, DT < AI < AT, and DT < AT < AI negative the supposition;

(5) outcomes DT ≠ AI = AT (DT < AI = AT, AI = AT < DT) negative the suppostions, especially discreaiting the belief that "mental cohesiveness" is the key influence on the product; and

(6) outcomes AI ≠ AT = DT (AI < AT = DT, AT = DT < AI) negative the supposition, especially discreaiting the belief that discipline methodology effectively molds a team into an individual.

Thus, interpreting the study's findings according to the basic suppositions consists of assessing how well the research conclusions have borne out the basic suppositions and how well the experimental evidence substantiates the general beliefs. On the whole, the study's findings soundly support the general beliefs presented above, although a few conclusions exist that are inconsistent with the basic suppositions or difficult to allay individually.

Support for the general beliefs was relatively stronger on process aspects than on product aspects, and in location comparisons rather than in dispersion comparisons. Overwhelming support came in the category of location comparisons on process aspects in which the research conclusions are distinguished by extremely low critical levels and by near unanimity with the basic supposition. In the category of dispersion comparisons on process aspects, only two outcomes indicated any distinction among the groups: one aspect supported the study's general beliefs and one aspect showed an explainable exception to them. Fairly strong support also came in the category of location comparisons on product aspects for which the only negative

evidence (besides the neutral AI = AT = DT conclusions) appeared in the form of several AI ≠ AT = DT conclusions. They indicate some areas in which the disciplined methodology was apparently ineffective in modifying a team's behavior toward that of an individual, probably due to a lack of fully developed training/experience with the methodology. Comparatively weaker support for the study's beliefs was recorded in the category of dispersion comparisons on product aspects. Although the basic suppositions were borne out in a number of the conclusions, there were also several distinctions of various forms which contravene the basic suppositions.

Thus, according to this interpretation, the study's findings strongly substantiate the claims that

(C1) methodological discipline is the key influence on the general efficiency of the software development process, and that

(C2) the disciplined methodology significantly reduces the material costs of software development.

The claims that

(C3) mental cohesiveness is the key influence on the general quality of the software development product, that

(C4) relative to an individual, an ad hoc team is mentally burdened by organizational overhead, and that

(C5) the disciplined methodology offsets the mental burden of organizational overhead and enables a team to behave more like an individual relative to the software product,

are moderately substantiated by the study's findings, with particularly mixed evidence for dispersion comparisons on product aspects.

It should be noted that there is a simpler, better-supported interpretive model for the location results alone. With the beliefs that a disciplined methodology provides for the minimum process cost and results in a product which in some aspects approximates the product of an individual and at worst approximates the product developed by an ad hoc team, the suppositions are DT $\leq$ AI,AT with respect to process and AI $\leq$ DT $\leq$ AT or AT $\leq$ DT $\leq$ AI with respect to product. The study's findings support these suppositions without exception.

According to Programming-Aspect Classification

It is desirable to examine the study's findings in view of the way that higher-level programming issues are reflected among the individual programming aspects. For this purpose, the aspects considered in this study were grouped into (so-called) programming aspect classes. Each class consists of aspects which are related by some common feature (for example, all aspects relating to the program's statements, statement types, statement nesting, etc.), and the classes are not necessarily disjoint (i.e., a given aspect may be included in two or more classes). A unique higher-level programming issue (in the example, control structure organization) is associated with each class.

The programming aspects of this study were organized into a hierarchy of nine aspect classes (with about 10% overlap overall), outlined as follows:

Higher-level Programming Issue:	Class:
Development Process Efficiency	
Effort (Job Steps)	I
Errors (Program Changes)	II
Final Product Quality	
Gross Size	III
Control-Construct Structure	IV
Data Variable Organization	V
Modularity	
Packaging Structure	VI
Invocation Organization	VII
Inter-Segment Communication	
Via Parameters	VIII
Via Global Variables	IX

The individual aspects comprising each class, together with the corresponding conclusions, are listed by classes in Tables 6.1 through 6.9. For each aspect class, it is interesting to jointly interpret the individual outcomes in an overall manner in order to see something of how these higher-level issues are affected by team size and methodological discipline.

Class I: Effort (Job Steps)

Within Class I (process aspects dealing with COMPUTER JOB STEPS), there is strong evidence of an important difference among the groups, in favor of the disciplined methodology, with respect to average development costs. As a class, these aspects directly reflect the frequency of computer system activities (i.e., module compilations and test program executions) during development. They are one possible way of measuring machine costs, in units of basic activities rather than monetary charges. Assuming that each computer system activity involves a certain expenditure of the programmer's time and effort (e.g., effective terminal contact, test result evaluation), these aspects indirectly reflect human costs of development (at least that portion exclusive of design work).

The strength of the evidence supporting a difference with respect to location comparisons within this class is

TABLE 6

Table 6.1 Conclusions for Class I, Effort (Job Steps)

programming aspect	location comparison outcome	location critical level	dispersion comparison outcome	dispersion critical level
COMPUTER JOB STEPS	DT < AI = AT	0.003	= =	
MODULE COMPILATION	DT < AI = AT	0.022	= =	
UNIQUE	DT < AI = AT	0.011	= =	
IDENTICAL	= =		= =	
PROGRAM EXECUTION	DT < AI = AT	0.022	= =	
MISCELLANEOUS	DT < AI = AT	0.144	AT = DT < AI	0.077
ESSENTIAL	DT < AI = AT	0.003	= =	
AVERAGE UNIQUE COMPILATIONS PER MODULE	DT < AI = AT	0.088	= =	
MAX. UNIQUE COMPILATIONS F.A.O. MODULE	DT < AI = AT	0.119	DT < AI < AT	0.051 &

MAX. is an abbreviation for MAXIMUM
F.A.O. is an abbreviation for FOR ANY ONE

alternative conclusions (from Table 5) showing relaxed differentiation:
(correspondence indicated via the & symbol)

		DT < AI = AT	0.003 &
		DT = AI < AT	0.051 &

TABLE 6

based on both (a) the near unanimity [8 out of 9 aspects] of the DT < AI = AT outcome and (b) the very low critical levels [<.025 for 5 aspects] involved. Indeed, the single exception among the location comparisons (AI = AT = DT on COMPUTER JOB STEPS\MODULE COMPILATIONS\IDENTICAL) is readily explained as a direct consequence of the fact that all teams made essentially similar usage (or nonuse, in this case, since identical compilations were not uncommon) of the on-line storage capability (for saving relocatable modules and thus avoiding identical recompilations). This was expected since all teams had been provided with identical storage capability, but without any training or urging to use it. The conclusions on location comparisons within this class are interpreted as demonstrating that

> employment of the disciplined methodology by a programming team reduces the average costs, both machine and human, of software development, relative to both individual programmers and programming teams not employing the methodology.

Examination of the raw data scores themselves indicates the magnitude of this reduction to be on the order of 2 to 1 (i.e., 50%) or better.

With respect to dispersion comparisons within this class, the evidence generally failed to make any distinctions among the groups [AI = AT = DT on 7 out of 9 aspects]. These null conclusions in dispersion comparisons are interpreted as demonstrating that

> variability of software development costs, especially machine costs, is relatively insensitive to programming team size and degree of methodological discipline.

The two exceptions on individual process aspects deserve mention. The COMPUTER JOB STEPS\MISCELLANEOUS aspect showed a AT = DT < AI dispersion distinction among the groups, reflecting the variability (as expected) of individual

programmers relative to programming teams in the area of building on-line tools to indirectly support software development (e.g., stand-alone module drivers, one-shot auxiliary computations, table generators, unanticipated debugging stubs, etc.). The MAX UNIQUE COMPILATIONS F.A.O. MODULE aspect showed a DT < AI = AT dispersion distinction among the groups at an extremely low critical level [<.005], reflecting the lower variation (increased predictability) of the disciplined teams relative to the ad hoc teams and individuals in terms of "worst case" compilation costs for any one module. The additional AI < AT distinction for this comparison is attributable to the fact that several teams in group AT built monolithic single-module systems, yielding rather inflated raw scores for this aspect.

Class II: Errors (Program Changes)

Within Class II (the process aspect PROGRAM CHANGES), there is strong evidence of an important difference among the groups, again in favor of the disciplined methodology, with respect to average number of errors encountered during implementation. Chapter V contains a detailed explanation of how program changes are counted. This aspect directly reflects the amount of textual revision to the source code during (postdesign) development. Claiming that textual revisions are generally necessitated by errors encountered while building, testing, and debugging software, independent research [Dunsmore and Gannon 77] has demonstrated a high (rank order) correlation between total program changes (as counted automatically according to a specific algorithm) and total error occurrences (as tabulated manually from exhaustive scrutiny of source code and test results) during software implementation. This aspect is thus a reasonable measure of the relative number of programming errors encountered outside of design work. Assuming that each

Table 6.2 Conclusions for Class II, Errors (Program Changes)

programming aspect	location comparison outcome	location :critical level	dispersion comparison outcome	dispersion :critical level
PROGRAM CHANGES	DT < AI < AT	: 0.184 &	≠ ≠	:

alternative conclusions (from Table 5) showing relaxed differentiation:
(correspondence indicated via the & symbol)

	DT < AI = AT : 0.003 &	
	DT = AI < AT : 0.184 &	

TABLE 6

textual revision involves an expenditure of programmer effort (e.g., planning the revision, on-line editing of source code), this aspect indirectly reflects the level of human effort devoted to implementation.

With respect to location comparison, the strength of the evidence supporting a difference among the groups is based on the very low critical level [<.005] for the DT < AI = AT outcome. The additional trend toward AI < AT is much less pronounced in the data. The interpretation is that

> the disciplined methodology effectively reduced the average number of errors encountered during software implementation.

This was expected since the methodology purposely emphasizes the criticality of the design phase and subjects the software design (code) to thorough reading and review prior to coding (key-in or testing), enhancing error detection and correction prior to implementation (testing).

With respect to dispersion comparison, no distinction among the groups was apparent, with the interpretation that

> variability in the number of errors encountered during implementation was essentially uniform across all three programming environments considered.

Class III: Gross Size

Within Class III (product aspects dealing with the gross size of the software at various hierarchical levels), there is evidence of certain consistent differences among the groups with respect to both average size and variability of size. As a class, these aspects directly reflect the number of objects and the average number of component (sub)objects per object, according to the hierarchical

TABLE 6

Table 6.3 Conclusions for Class III. Gross Size

programming aspect	location comparison outcome	location critical level	dispersion comparison outcome	dispersion critical level
MODULES AVERAGE SEGMENTS PER MODULE AVERAGE GLOBAL VARIABLES PER MODULE	= = =		= DT = AI < AT =	0.021
SEGMENTS	AI < AT = DT	0.063	=	
AVERAGE STATEMENTS PER SEGMENT	AT = DT < AI	0.170	=	
AVERAGE NONGLOBAL VARIABLES PER SEGMENT	=		=	
PARAMETER	AI < AT = DT	0.174	=	
LOCAL	=		=	
DATA VARIABLES	AI < AT = DT	0.069	=	
DATA VARIABLE SCOPE COUNTS \ GLOBAL	AI < AT = DT	0.147	AI = AT < DT	0.124
DATA VARIABLE SCOPE COUNTS \ NONGLOBAL	=		=	
PARAMETER	AI < AT = DT	0.127	AI = AT < DT	0.106
LOCAL	=		=	
LINES	AI < DT < AT	0.119 &	=	
STATEMENTS	=		AT < DT = AI	0.195
AVERAGE TOKENS PER STATEMENT	=		AI = AT < DT	0.106
TOKENS	=		=	

alternative conclusions (from Table 5) showing relaxed differentiation:
(correspondence indicated via the & symbol)

	DT = AI < AT	0.061 &	
	AI < AT = DT	0.113 &	

organization (imposed by the programming language) of software into objects such as modules, segments, data variables, lines, statements, and tokens.

With respect to location comparisons within this class, the non-null conclusions [7 out of 17 aspects] are nearly unanimous [5 out of 7] in the AI < AT = DT outcome. The interpretation is that individuals tend to produce software which is smaller (in certain ways) on the average than that produced by teams. It is unclear whether such spareness of expression, primarily in segments, global variables, and formal parameters, is advantageous or not. The two non-null exceptions to this AI < AT = DT trend deserve mention, since the one is only nominally exceptional and actually supportive of the tendency upon closer inspection, while the other indicates a size aspect in which the disciplined methodology enabled programming teams to break out of the pattern of distinction from individual programmers. The AT = DT < AI outcome on AVERAGE STATEMENTS PER SEGMENT is a simple consequence of the outcome for the number of STATEMENTS [AI = AT = DT] and the outcome for the number of SEGMENTS [AI < AT = DT] and it still fits the overall pattern of AI ≠ AT = DT on location differences on size aspects. On the LINES aspect, the DT = AI < AT distinction breaks the pattern since DT is associated with AI and not with AT. Since the number of statements was roughly the same for all three groups, this difference must be due mainly to the stylistic manner of arranging the source code (which was free-format with respect to line boundaries), to the amount of documentation comments within the source code, and to the number of lines taken up in data variable declarations.

With respect to dispersion comparisons within this class, the few aspects which do indicate any distinction

among the groups [5 out of 17 aspects] seem to concur on the AI = AT < DT outcome. This pattern, which associates increased variation in certain size aspects with the disciplined methodology, is somewhat surprising and lacks an intuitive explanation in terms of the experimental treatments. The exception DT = AI < AT on AVERAGE SEGMENTS PER MODULE is really an exaggeration due to the fact of several AT teams implementing monolithic single-module systems, as mentioned above. The exception AT < DT = AI on STATEMENTS is only a very slight trend, reflecting the fact that the AT products rather consistently contained the largest numbers of statements.

One overall observation for Class III is that while certain distinctions did consistently appear (especially for location but also for dispersion comparisons) at the middle levels of the hierarchical scale (segments, data variables, lines, and statements), no distinctions appeared at either the highest (modules) or lowest (tokens) levels of size. The null conclusions for size in modules and average module size seem attributable to the fact that particular programming tasks or application domains often have standard designs at the topmost conceptual levels which strongly influence the organization of software systems at this highest level of gross size. In this case, the symbol-table/scanning/parsing/code-generation design is extremely common for language translation problems (i.e., compilers), regardless of the particular parsing technique or symbol table organization employed, and the modules of nearly every system in the study directly reflected this common design. The null conclusions for size in tokens is interpretable in view of Halstead's software science concepts [Halstead 77], according to which the program length N is predictable from the number of basic input-output parameters η_2^* and the language level λ. Since the functional specification,

application area, and implementation language were all fixed in this study, both η_2^* and λ should be constant for each of the software systems, implying virtually constant program lengths N. Since program length N can be regarded as roughly equivalent to the number of tokens in a program, the study's data seem to support the software science concepts in this instance.

Class IV: Control-Construct Structure

Within Class IV (product aspects dealing with the software's organization according to statements, constructs, and control structures), there are only a few distinctions made between the groups.

With respect to location comparisons, the few [5 out of 24] aspects that showed any distinction at all were unanimous in concluding DT = AI < AT. Essentially, three particular issues were involved. The STATEMENT TYPE COUNTS\IF, STATEMENT TYPE PERCENTAGES\IF, and DECISIONS aspects are all related to the frequency of programmer-coded decisions in the software product. Their common outcome DT = AI < AT is interpreted as demonstrating an important area in which the disciplined methodology causes a programming team to behave like an individual programmer. The number of decisions has been commonly accepted, and even formalized [McCabe 76], as a measure of program complexity since more decisions create more paths through the code. Thus, the disciplined methodology effectively reduced the average complexity from what it otherwise would have been. The STATEMENT TYPE COUNTS\RETURN aspect indicates a difference between the ad hoc teams and the other two groups. Since the EXIT and RETURN statements are restricted forms of GOTOs, this difference seems to hint at another area in which the disciplined methodology improves conceptual

Table 6.4 Conclusions for Class IV, Control-Construct Structure

programming aspect	location comparison outcome	location critical level	dispersion comparison outcome	dispersion critical level
STATEMENTS	= =		AT < DT = AI	0.195
STATEMENT TYPE COUNTS :				
:=	= =		= =	
IF	DT = AI < AT	0.078	= =	
CASE	= =		= =	
WHILE	= =		= =	
EXIT	= =		= =	
(PROC)CALL	= =		DT < AI = AT	0.032
NONINTRINSIC	= =		DT < AI = AT	0.186
INTRINSIC	DT = AI < AT	0.173	= =	
RETURN	DT = AI < AT	0.086	DT < AI < AT	0.139 &
STATEMENT TYPE PERCENTAGES :				
:=	= =		= =	
IF	DT = AI < AT	0.106	= =	
CASE	= =		= =	
WHILE	= =		= =	
EXIT	= =		= =	
(PROC)CALL	= =		= =	
NONINTRINSIC	= =		= =	
INTRINSIC	= =		= =	
RETURN	= =		DT = AI < AT	0.040
AVERAGE STATEMENT NESTING LEVEL	= =		= =	
DECISIONS	DT = AI < AT	0.146	= =	
FUNCTION CALLS	= =		= =	
NONINTRINSIC	= =		= =	
INTRINSIC	= =		= =	

alternative conclusions (from Table 5) showing relaxed differentiation:
(correspondence indicated via the & symbol)

		DT = AI < AT	0.003 &
		DT < AI = AT	0.139 &

TABLE 6

control over program structure. The STATEMENT TYPE COUNTS\(PROC)CALL\INTRINSIC aspect also indicates a slight trend in the area of the frequency of input-output operations, which seems interpretable only as a result of stylistic differences.

With respect to dispersion comparisons, only two particular issues were involved. The STATEMENT TYPE COUNTS\RETURN and STATEMENT TYPE PERCENTAGE\RETURN aspects both indicated a strong DT = AI < AT difference, suggesting that the frequency of these restricted GOTOs is an area in which the disciplined methodology reduces variability, causing a programming team to behave more like an individual programmer. The STATEMENT TYPE COUNTS\(PROC)CALL and STATEMENT TYPE COUNTS\(PROC)CALL\NONINTRINSIC aspects both showed a DT < AI = AT distinction among the groups, which is dealt with more appropriately within Class VII below.

In summary of Class IV, the interpretation is that the functional component of control-construct organization is largely unaffected by team size and methodological discipline, probably due to the overriding effect of project/task uniformity/commonality. However, two facets of the control component that were influenced were the frequency of decisions (especially IF statements) and the frequency of restricted GOTOs (especially RETURN statements). For these aspects, the disciplined methodology seems to have altered the size of the program's control structure (and reduced its complexity) from that of a team's product to that of an individual's product.

Class V: Data Variable Organization

Within Class V (product aspects dealing with data variables and their organization within the software), there

Table 6.5 Conclusions for Class V, Data Variable Organization

programming aspect	location comparison outcome	location critical level	dispersion comparison outcome	dispersion critical level
DATA VARIABLES	AI < AT = DT	0.069	= =	
DATA VARIABLE SCOPE COUNTS :				
GLOBAL	AI < AT = DT	0.147	AI = AT < DT	0.124
ENTRY	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
NONENTRY	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
MODIFIED	AI < AT = DT	0.161	= =	
UNMODIFIED	= =		= =	
NONGLOBAL	= =		= =	
PARAMETER	AI < AT = DT	0.127	AI = AT < DT	0.106
VALUE	= =		= =	
REFERENCE	= =		AI < AT = DT	0.019
LOCAL	= =		= =	
DATA VARIABLE SCOPE PERCENTAGES :				
GLOBAL	= =		AI = AT < DT	0.075
ENTRY	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
NONENTRY	= =		= =	
MODIFIED	= =		DT < AI = AT	0.021
UNMODIFIED	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
NONGLOBAL	= =		AI = AT < DT	0.075
PARAMETER	AI < AT = DT	0.150	AI = AT < DT	0.055
VALUE	= =		AI = AT < DT	0.094
REFERENCE	= =		AI < AT = DT	0.152
LOCAL	AT = DT < AI	0.109	= =	
AVERAGE GLOBAL VARIABLES PER MODULE	= =		= =	
ENTRY	= =		= =	
NONENTRY	= =		= =	
MODIFIED	= =		DT = AI < AT	0.110
UNMODIFIED	= =		= =	
AVERAGE NONGLOBAL VARIABLES PER SEGMENT	= =		= =	
PARAMETER	AI < AT = DT	0.174	= =	
LOCAL	= =		= =	

TABLE 6

are several distinctions among the groups, with an overall trend for both the location and dispersion comparisons. Data variable organization was, however, not emphasized in the disciplined methodology, nor in the academic course which the participants in group DT were taking. With respect to location comparisons, all aspects showing any distinction at all were unanimous in concluding AI ≠ AT = DT. The trend for individuals to differ from teams, regardless of the disciplined methodology, appears not only for the total number of data variables declared, but also for data variables at each scope level (global, parameter, local) in one fashion or another. The difference regarding formal parameters is especially prominent, since it shows up for their raw count frequency, their normalized percentage frequency, and their average frequency per natural enclosure (segment). With respect to dispersion comparisons, the apparent overall trend for aspects which show a distinction is toward the AI = AT < DT outcome. No particular interpretation in view of the experimental treatments seems appropriate. Exceptions to this trend appeared for both the raw count and percentage of call-by-reference paramenters (both AI < AT = DT), as well as two other aspects.

Class VI: Packaging Structure

Within Class VI (product aspects dealing with modularity in terms of the packaging structure), there are essentially no distinctions among the groups, except for two location comparison issues. Most of the aspects in this class are also members of Class III, Gross Size, but are (re)considered here to focus attention upon the packaging characteristics of modularity (i.e., how the source code is divided into modules and segments, what type of segments, etc.). The disciplined methodology did not explicitly

TABLE 6

Table 6.6 Conclusions for Class VI, Packaging Structure

programming aspect	location comparison outcome	location critical level	dispersion comparison outcome	dispersion critical level
MODULES	= =		= =	
AVERAGE SEGMENTS PER MODULE AVERAGE GLOBAL VARIABLES PER MODULE	= = = =		DT = AI < AT = =	0.021
SEGMENTS SEGMENT TYPE COUNTS \ FUNCTION SEGMENT TYPE COUNTS \ PROCEDURE SEGMENT TYPE PERCENTAGES \ FUNCTION SEGMENT TYPE PERCENTAGES \ PROCEDURE	AI < AT = DT = = = =	0.063	= = = = =	
AVERAGE STATEMENTS PER SEGMENT AVERAGE NONGLOBAL VARIABLES PER SEGMENT PARAMETER LOCAL	AT = DT < AI = AI < AT = DT =	0.170 0.174	= = = =	

include (nor did group DT's course work cover) concepts of modularization or criteria for evaluating good modularity; hence, no particular distinctions among the groups were expected in this area (Classes VI and VII).

With respect to location comparisons, the AI < AT = DT outcome for the SEGMENTS aspects, along with the companion outcome AT = DT < AI for the AVERAGE STATEMENTS PER SEGMENT aspect (as explained under Class III above), indicates one area of packaging that is apparently sensitive to team size. Individual programmers built the system with fewer, but larger (on the average), segments than either the ad hoc teams or the disciplined teams. The AI < AT = DT outcome for the AVERAGE NONGLOBAL VARIABLES PER SEGMENT\PARAMETER aspect indicates that average "calling sequence" length, curiously enough, is another area of packaging sensitive to team size. With respect to dispersion comparisons, there really were no differences, since the single non-null outcome for AVERAGE SEGMENTS PER MODULE is actually a fluke (raw scores for AT are exaggerated by the several monolithic systems) as explained above. The overall interpretation for this class is that

> modularity, in the sense of packaging code into segments and modules, is essentially unaffected by team size or methodological discipline, except for a tendency by individual programmers toward fewer, longer segments than programming teams.

Class VII: Invocation Organization

Within Class VII (product aspects dealing with modularity in terms of the invocation structure), there are two distinction trends for location comparisons, but no clear pattern for the dispersion comparison conclusions. This class consists of raw counts and average-per-segment

Table 6.7 Conclusions for Class VII, Invocation Organization

programming aspect	location comparison outcome	location critical level	dispersion comparison outcome	dispersion critical level
INVOCATIONS			AT = DT < AI	0.020
FUNCTION				
NONINTRINSIC				
INTRINSIC				
PROCEDURE			DT < AI = AT	0.032
NONINTRINSIC			DT < AI = AT	0.186
INTRINSIC	DT = AI < AT	0.173		
NONINTRINSIC			AT = DT < AI	0.051
INTRINSIC	DT = AI < AT	0.043		
AVG. INVOCATIONS PER (CALLING) SEGMENT				
FUNCTION				
NONINTRINSIC				
INTRINSIC				
PROCEDURE				
NONINTRINSIC				
INTRINSIC			DT < AI = AT	0.065
NONINTRINSIC	AT = DT < AI	0.169		
INTRINSIC				
AVG. INVOCATIONS PER (CALLED) SEGMENT	AT = DT < AI	0.169		
FUNCTION	AT = DT < AI	0.193	AT < DT = AI	0.141
PROCEDURE				

AVG. is an abbreviation for AVERAGE

frequencies for invocations (procedure CALL statements or function references in expressions) and is considered separately from the previous class since modularity involves not only the manner in which the system is packaged, but also the frequency with which the pieces might be invoked. For the raw count frequencies of calls to intrinsic procedures and intrinsic routines, the trend is for the individuals and disciplined teams to exhibit fewer calls than the ad hoc teams. These intrinsic procedures are almost exclusively the input-output operations of the language, while the intrinsic functions are mainly data type conversion routines. The second trend for location comparisons occurs for two aspects (a third aspect is actually redundant) related to the average frequency of calls to programmer-defined routines, in which the individuals display higher average frequency than either type of team. This seems coupled with group AI's preference for fewer but larger routines, as noted above. With respect to dispersion comparisons, several distinctions appear within this class, but no overall interpretation is readily apparent (except for a consistent reflection of a DT < AI difference, with AT falling in between, leaning one side or the other).

Class VIII: Inter-Segment Communication via Parameters

Within Class VIII (product aspects dealing with inter-segment communication via formal parameters), there are only a few distinctions among the groups. With respect to location comparisons, the total frequency of parameters and the average frequency of parameters per segment both show a difference. The interpretation is that

> the individual programmers tend to incorporate less inter-segment communication via parameters, on the average, than either the ad hoc or the disciplined

TABLE 6

Table 6.8 Conclusions for Class VIII, Communication via Parameters

programming aspect	location comparison outcome	location critical level	dispersion comparison outcome	dispersion critical level
DATA VARIABLE SCOPE COUNTS\NONGLOBAL :				
PARAMETER	AI < AT = DT	0.127	AI = AT < DT	0.106
VALUE	= =		= =	
REFERENCE	= =		AI < AT = DT	0.019
AVG. NONGLOBAL VARIABLES PER SEGMENT :				
PARAMETER	AI < AT = DT	0.176	= =	
PARAMETER PASSAGE TYPE PERCENTAGES :				
VALUE	= =		AI < AT = DT	0.160
REFERENCE	= =		AI < AT = DT	0.160

AVG. is an abbreviation for AVERAGE

TABLE 6

programming teams.

With respect to dispersion comparisons, in addition to the difference in the raw count of parameters referred to in Class V, there is a strong difference in the variability of the number of call-by-reference parameters, also apparent in the percentages-by-type-of parameter aspects. The interpretation is that

> the individual programmers were more consistent as a group in their use (in this case, avoidance) of reference parameters than either type of programming team.

Class IX: Inter-Segment Communication via Global Variables

Within Class IX (product aspects dealing with inter-segment communication via global variables), there are several differences among the groups, including two which indicate the beneficial influence of the disciplined methodology. This class is composed of aspects dealing with absolute frequency of globals, average frequency of globals per module, segment-global usage pairs (frequency of access paths from segments to globals), and segment-global-segment data bindings (frequency of communication paths between segments via global variables).

With respect to location comparisons, there is the AI < AT = DT distinction in sheer numbers of globals, particularly globals which are modified during execution, as noted in Class V. However, when averaged per module, there appears to be no distinction in the frequency of globals. The AI < AT = DT difference in the number of possible segment-global access paths makes sense as the result of group AI having both fewer segments and fewer globals. All three groups had essentially similar average levels of actual segment-global access paths, but several differences

Table 6.9 Conclusions for Class IX, Communication via Global Variables

programming aspect	location comparison outcome	location critical level	dispersion comparison outcome	dispersion critical level
DATA VARIABLE SCOPE COUNTS \ GLOBAL	AI < AT = DT	0.147	AI = AT < DT	0.124
ENTRY				
MODIFIED				
UNMODIFIED				
NONENTRY				
MODIFIED				
UNMODIFIED				
MODIFIED	AI < AT = DT	0.161		
UNMODIFIED				
AVERAGE GLOBAL VARIABLES PER MODULE				
ENTRY				
NONENTRY				
MODIFIED			DT = AI < AT	0.110
UNMODIFIED				
(SEGMENT,GLOBAL) ACTUAL USAGE PAIRS				
ENTRY				
MODIFIED				
UNMODIFIED				
NONENTRY				
MODIFIED				
UNMODIFIED				
MODIFIED			AT < DT = AI	0.106
UNMODIFIED				
(SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS	AI < AT = DT	0.122	AI < DT < AT	0.052 &
ENTRY				
MODIFIED				
UNMODIFIED				
NONENTRY			AI < AT = DT	0.078
MODIFIED			DT = AI < AT	0.051
UNMODIFIED			AI < DT < AT	0.172 #
MODIFIED				
UNMODIFIED				
(SEGMENT,GLOBAL) USAGE PAIR REL.PERCENT.				
ENTRY	AT < DT < AI	0.117 #		
MODIFIED	AT < DT < AI	0.125 $		
UNMODIFIED				
NONENTRY				
MODIFIED				
UNMODIFIED	AT < DT = AI	0.154		
MODIFIED				
UNMODIFIED				

TABLE 6

(SEGMENT,GLOBAL,SEGMENT) DATA BINDINGS :									
ACTUAL									
SUBFUNCTIONAL INDEPENDENT									
POSSIBLE RELATIVE PERCENTAGE	DT = AI < AT	:	0.186			AI < AT = DT	:	0.126	
						DT = AI < AT	:	0.152	

REL.PERCENT. is an abbreviation for RELATIVE PERCENTAGE

alternative conclusions (from Table 5) showing relaxed differentiation:
(correspondence indicated via the #, @, &, and $ symbols)

				AI < AT = DT	:	0.020 &
				DT = AI < AT	:	0.051 &
	AT < DT = AI	:	0.082 #			
	AT = DT < AI	:	0.111 #			
				AI < AT = DT	:	0.116 @
				DT = AI < AT	:	0.156 @
	AT < DT = AI	:	0.113 $			
	AT = DT < AI	:	0.113 $			

TABLE 6

appear in the relative percentage (actual-to-possible ratio) category. These three instances of AT < DT = AI differences indicate that the degree of "globality" for global variables was higher for the individuals and the disciplined teams than for the ad hoc teams. Finally, another AT ≠ DT = AI difference appears for the frequency of possible segment-global-segment data bindings, indicating a positive effect of the disciplined methodology in reducing the possible data coupling among segments. It may be noted that these last two categories of aspects, segment-global usage relative percentages and segment-global-segment data bindings, also reflect upon the quality of modularization, since good modularity should promote the degree of "globality" for globals and minimize the data coupling among segments. The interpretation here is that

> certain aspects of inter-segment communication via globals seems to be positively influenced, on the average, by the disciplined methodology.

with respect to dispersion comparisons, there is a diversity of differences in this class, without any unifying interpretation in terms of the experimental treatments.

Miscellaneous

The cyclomatic complexity and software science metrics, whose results have not been integrated into the two interpretive frameworks discussed above, definitely merit some interpretation.

On location comparisons, the results for cyclomatic complexity measures exhibited a common underlying trend, namely, DT ≤ AT ≤ AI. In fact, the non-null outcomes were usually either AT = DT < AI or else DT < AI = AT. This says that either the teams were differentiated from the

individuals or else the disciplined methodology was differentiated from the ad hoc approach, depending on the particular variation of cyclomatic complexity involved. This corresponds well with the intuition that team programming alone should force a general reduction of cyclomatic complexity for individual routines, and that use of the disciplined methodology within team programming should promote this effect even further. The observed results for the cyclomatic complexity metrics seem to display this kind of behavior.

The generally weaker differentiation (i.e., larger critical levels) observed for the cyclomatic complexity aspects relative to other aspects considered in the study is quite understandable in light of the fact that all 19 systems were coded in a structured-programming language which greatly restricts potential control flow patterns. We would expect cyclomatic complexity metrics to be more useful in the context of unrestrictive programming languages such as Fortran.

The results for software science quantities are somewhat disappointing: surprisingly few distinctions among the groups were obtained. On location comparisons, only the vocabulary and estimated length metrics (the latter is a function solely of the former) yielded non-null conclusions. Their AI < AT = DT outcome corresponds to that obtained for the number of segments and data variables, both of which contribute heavily to the number of "operator/operands." The overall interpretation here is that the software science metrics appear to be insensitive to differences in how software is developed. Maybe these measures, with their actuarial nature, are sensitive only to gross factors in software development (e.g., project, application area, implementation language), all of which were held constant in

this experiment.

CHAPTER IX

IX. SUMMARY AND CONCLUSIONS

A practical methodology was designed and developed for experimentally and quantitatively investigating software development phenomena. It was employed to compare three particular software development environments and to evaluate the relative impact of a particular disciplined methodology (made up of so-called modern programming practices). The experiments were successful in measuring differences among programming environments and the results support the claim that disciplined methodology effectively improves both the process and product of software development.

One way to substantiate the claim for improved process is to measure the effectiveness of the particular programming methodology via the number of bugs initially in the system (i.e., in the initial source code) and the amount of effort required to remove them. (This criteria has been suggested independently by Professor M. Shooman of Polytechnic Institute of New York [Shooman 78].) Although neither of these measures was directly computed, they are each closely associated with one of the process aspects considered in the study: PROGRAM CHANGES and COMPUTER JOB STEPS\ESSENTIAL, respectively. The location comparison statistical conclusions for both these aspects affirmed DT < AI = AT outcomes at very low (<.01) significance levels, indicating that on the average the disciplined teams measured lower than either the ad hoc individuals or the ad hoc teams which both measured about the same. Thus, the evidence collected in this study strongly confirms the effectiveness of the disciplined methodology in building reliable software efficiently.

The second claim, that the product of a disciplined

team should closely resemble that of a single individual since the disciplined methodology assures a semblance of conceptual integrity within a programming team, was partially substantiated. In many product aspects the products developed using the disciplined methodology were either similar to or tended toward the products developed by the individuals. In no case did any of the measures show the disciplined teams' products to be worse than those developed by the ad hoc teams. It is felt that the superficiality of most of the product measures was chiefly responsible for the lack of stronger support for this second claim. The need for product measures with increased sensitivity to critical characteristics of software is very clear.

The results of these experiments will be used to guide further experiments and will act as a basis for analysis of software development products and processes in the Software Engineering Laboratory at NASA's Goddard Space Flight Center [Basili et al. 77]. The intention is to pursue this type of empirical research, especially extending the study to more sophisticated and promising software metrics.

Appendix 1. Statistical Description of Raw Scores

Across-all-groups and within-each-group sample mean values and standard deviations supply a statistical description of the raw scores obtained in the experiment for each programming aspect.
N.B. The parenthesized numbers refer to the explanatory notes in Chapter IV. The asterisks mark confirmatory aspects; exploratory aspects are unmarked.

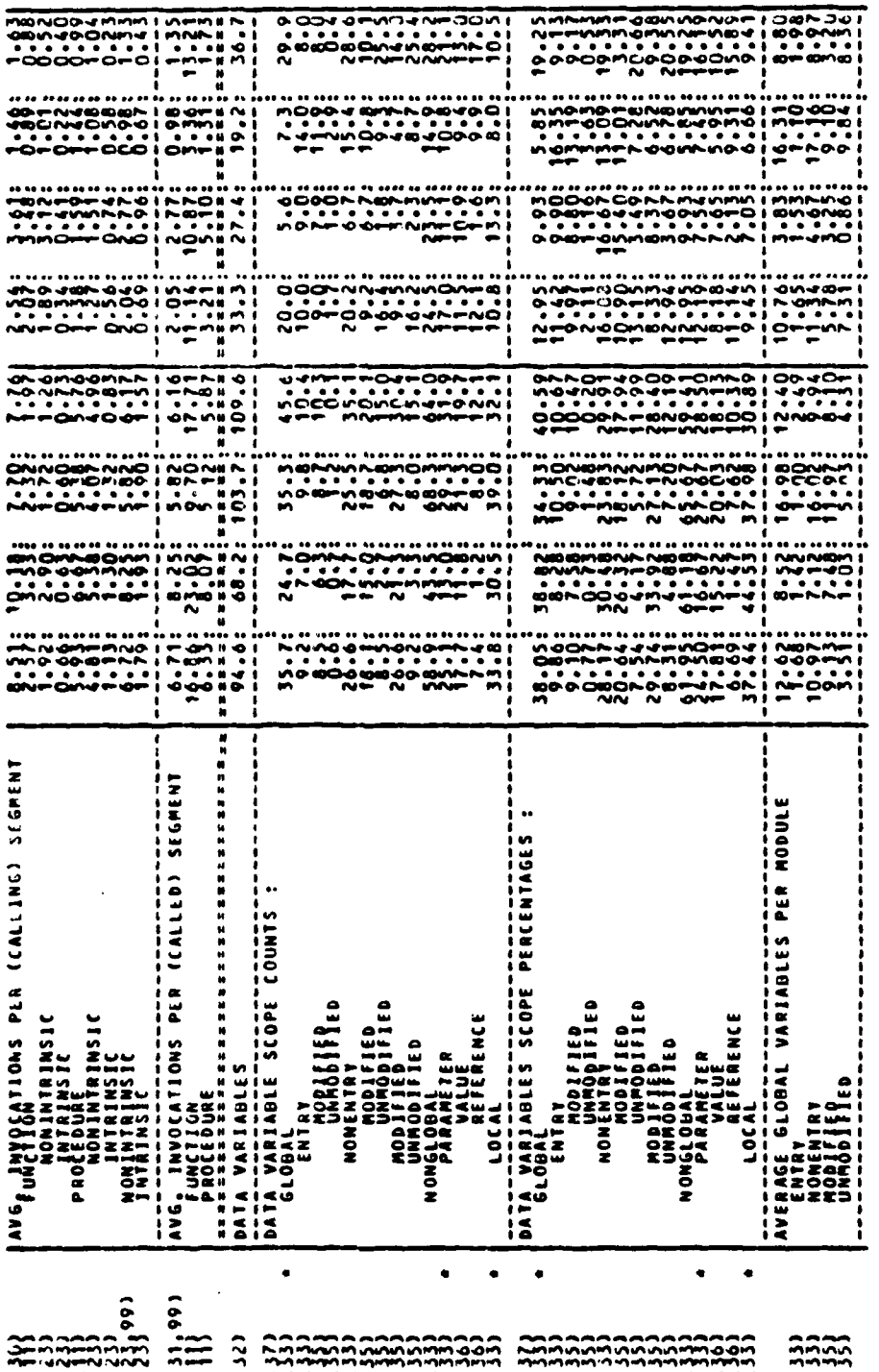
		programming aspects	mean values all	mean values AI	mean values AT	mean values DT	standard deviations all	standard deviations AI	standard deviations AT	standard deviations DT
		rudimentary process aspects								
(1)	*	COMPUTER JOB STEPS	157.0	185.5	223.5	75.6	93.8	90.7	84.2	25.5
(2)	*	MODULE COMPILATION	92.5	102.2	136.5	46.6	52.1	47.3	43.9	15.2
	*	UNIQUE	73.1	81.5	108.5	35.9	41.4	32.5	37.9	11.6
		IDENTICAL	19.4	20.8	28.2	10.7	17.9	22.8	19.7	6.1
(4)	*	PROGRAM EXECUTION	60.3	76.0	82.5	28.0	44.6	43.8	51.5	14.5
(5)		MISCELLANEOUS	4.2	7.5	4.7	1.0	5.5	7.6	4.9	1.5
(6)	*	ESSENTIAL	133.4	157.3	190.7	63.9	81.4	71.7	81.4	24.2
		AVERAGE UNIQUE COMPILATIONS PER MODULE	31.74	28.97	59.80	10.07	40.69	17.31	63.37	5.29
(8)		MAX. UNIQUE COMPILATIONS F.A.O. MODULE	51.0	56.7	81.5	20.0	44.5	33.6	58.0	9.9

MAX. is an abbreviation for MAXIMUM
F.A.O. is an abbreviation for FOR ANY ONE

		elaborative process aspects								
(9)	*	PROGRAM CHANGES	335.2	353.0	522.7	159.1	237.9	145.0	304.8	56.2
		rudimentary product aspects								
(10)	*	MODULES	4.5	3.8	5.3	4.3	3.7	3.1	5.6	2.1
(11)	*	SEGMENTS	40.1	30.7	47.7	41.7	12.1	9.5	13.4	7.9
(12)		SEGMENT TYPE COUNTS :								
(13)		FUNCTION	6.2	4.7	8.8	5.3	5.2	5.7	5.1	4.8
(14)		PROCEDURE	33.9	26.0	38.8	36.4	13.2	13.0	15.0	10.0
(12)		SEGMENT TYPE PERCENTAGES :								

(11)		FUNCTION	17.14	19.05	19.68	13.33	18.22	28.43	12.83	12.57
(11)		PROCEDURE	82.86	80.95	80.32	86.67	18.22	28.43	12.83	12.57
		AVERAGE SEGMENTS PER MODULE	14.62	10.53	21.30	12.39	11.40	4.62	17.29	7.46
(14)	*	LINES	1323.5	1026.7	1676.5	1275.3	409.6	330.8	399.6	252.2
(15)	*	STATEMENTS	609.6	563.3	674.2	593.9	116.0	136.7	70.7	118.3
(16)		STATEMENT TYPE COUNTS :								
		:=	205.6	202.3	204.3	209.4	63.3	53.4	44.6	89.1
	*	IF	78.7	68.2	102.5	67.0	28.6	31.5	18.0	21.1
	*	CASE	6.6	7.0	5.2	7.4	3.6	3.7	3.0	4.1
	*	WHILE	25.2	25.2	27.2	23.6	5.6	6.7	5.2	5.3
	*	EXIT	2.7	4.7	3.2	0.7	4.6	5.8	5.4	1.3
(99)		(PROC)CALL	230.0	205.5	249.5	236.4	56.5	81.3	59.0	12.4
(23,99)		NONINTRINSIC	187.0	167.3	192.3	199.3	54.2	74.8	64.2	10.5
(23,99)		INTRINSIC	43.0	38.2	57.0	35.1	18.8	19.3	17.9	13.9
	*	RETURN	60.7	50.3	82.2	51.3	22.6	9.0	30.0	6.5
(16)		STATEMENT TYPE PERCENTAGES :								
		:=	33.78	36.72	30.50	34.09	8.03	8.81	7.17	8.16
	*	IF	12.56	11.56	15.22	11.16	3.20	3.62	2.15	2.37
	*	CASE	1.10	1.32	0.77	1.20	0.52	0.68	0.41	0.40
	*	WHILE	4.19	4.58	4.05	4.06	0.75	0.64	0.74	0.88
	*	EXIT	0.40	0.70	0.42	0.13	0.61	0.78	0.68	0.22
		(PROC)CALL	38.00	36.02	36.97	40.59	7.48	7.97	8.08	6.91
(23)		NONINTRINSIC	31.09	29.58	28.35	34.74	8.41	9.60	8.27	7.27
(23)		INTRINSIC	6.91	6.53	8.62	5.84	2.56	2.59	2.88	1.65
	*	RETURN	9.99	9.23	12.07	8.86	2.80	1.90	3.80	1.45
	*	AVERAGE STATEMENTS PER SEGMENT	15.97	19.03	14.70	14.43	3.89	4.62	2.79	2.71
(26)	*	AVERAGE STATEMENT LIST NESTING LEVEL	2.596	2.760	2.573	2.476	0.268	0.222	0.336	0.189
(27)	*	DECISIONS	110.5	100.3	135.2	98.0	31.5	36.7	18.5	26.0
(99)		FUNCTION CALLS	90.8	90.2	104.8	79.4	44.1	65.2	33.8	31.6
(23,99)		NONINTRINSIC	65.8	72.8	76.5	50.7	40.6	59.1	38.4	19.2
(23,99)		INTRINSIC	25.0	17.3	28.3	28.7	13.5	9.0	15.3	14.1
(28)	*	TOKENS	3340.2	3072.8	3707.0	3255.0	797.4	812.2	477.4	976.4
(28)	*	AVERAGE TOKENS PER STATEMENT	5.45	5.45	5.50	5.40	0.55	0.37	0.43	0.80
(29)		INVOCATIONS	320.8	295.7	354.2	313.9	54.1	73.6	44.4	28.5
(11,99)		FUNCTION	90.8	90.2	104.8	79.4	44.1	65.2	33.8	31.6
(23,99)		NONINTRINSIC	65.8	72.8	76.5	50.7	40.6	59.1	38.4	19.2
(23,99)		INTRINSIC	25.0	17.3	28.3	28.7	13.5	9.0	15.3	14.1
(11,99)		PROCEDURE	230.0	205.5	249.5	234.4	56.5	81.3	59.0	12.4
(23,99)		NONINTRINSIC	187.0	167.3	192.3	199.3	54.2	74.8	64.2	10.5
(23,99)		INTRINSIC	43.0	38.2	57.0	35.1	18.8	19.3	17.9	13.9
(23)		NONINTRINSIC	252.8	240.2	268.8	250.0	43.0	61.3	43.7	19.7
(23)		INTRINSIC	68.0	55.5	85.5	63.9	21.0	18.3	17.7	17.4

APPENDIX 1



	AVERAGE NONGLOBAL VARIABLES PER SEGMENT	14.74:	14.38:	14.70:	15.07	4.93:	6.72:	3.10:	5.19
(33)	PARAMETER	5.90:	3.97:	6.23:	7.27	3.27:	2.26:	2.20:	4.22
(33)	LOCAL	8.84:	10.40:	8.48:	7.80	3.39:	5.14:	2.10:	2.22
(39)	PARAMETER PASSAGE TYPE PERCENTAGES :								
(36)	VALUE	81.12:	91.23:	76.73:	76.21	25.21:	14.49:	25.33:	32.36
(36)	REFERENCE	18.88:	8.77:	23.27:	23.79	25.21:	14.49:	25.33:	32.36
(40)	(SEGMENT,GLOBAL) ACTUAL USAGE PAIRS	142.5:	120.7:	151.3:	153.7	36.1:	35.4:	19.8:	42.7
(33)	ENTRY	55.3:	48.3:	53.3:	65.0	53.7:	53.6:	74.4:	39.3
(35)	MODIFIED	54.8:	48.3:	52.3:	62.4	52.6:	51.5:	72.9:	39.4
(35)	UNMODIFIED	0.8:	0.8:	1.0:	0.6	1.7:	1.3:	6.4:	1.5
(33)	NONENTRY	86.9:	71.5:	98.0:	90.7	57.2:	47.8:	66.4:	62.6
(35)	MODIFIED	75.4:	67.0:	90.2:	69.9	54.1:	44.8:	61.8:	60.0
(35)	UNMODIFIED	11.6:	4.5:	7.8:	20.9	26.6:	4.2:	10.1:	43.1
(35)	MODIFIED	130.2:	115.3:	142.5:	132.3	32.2:	32.6:	17.5:	40.0
(35)	UNMODIFIED	12.4:	5.3:	8.8:	21.4	26.3:	4.2:	9.4:	42.8
(40)	(SEGMENT,GLOBAL) POSSIBLE USAGE PAIRS	828.1:	488.3:	1195.3:	804.4	518.2:	182.9:	680.4:	375.5
(33)	ENTRY	269.4:	170.7:	368.3:	269.3	394.0:	234.1:	646.7:	227.8
(35)	MODIFIED	262.6:	163.5:	355.5:	267.9	390.6:	219.7:	645.0:	228.8
(35)	UNMODIFIED	6.8:	7.2:	12.8:	1.4	19.4:	16.1:	31.4:	3.8
(33)	NONENTRY	558.6:	317.7:	827.0:	535.1	535.6:	225.6:	779.5:	425.4
(35)	MODIFIED	397.7:	277.8:	563.3:	358.6	362.9:	182.5:	474.2:	369.3
(35)	UNMODIFIED	160.9:	39.8:	263.7:	176.6	302.8:	50.4:	434.8:	299.8
(35)	MODIFIED	660.3:	441.3:	918.8:	626.4	379.3:	139.9:	472.9:	335.6
(35)	UNMODIFIED	167.7:	47.0:	276.5:	178.0	299.7:	50.5:	426.5:	298.9
(40) *	(SEGMENT,GLOBAL) USAGE PAIR REL.PERCENT.	22.07:	26.38:	16.80:	22.90	9.59:	7.49:	9.50:	10.32
(33)	ENTRY	28.97:	43.23:	13.83:	29.71	18.99:	17.55:	13.39:	15.57
(35)	MODIFIED	29.26:	43.35:	14.57:	29.77	19.03:	17.32:	14.57:	15.64
(35)	UNMODIFIED	6.42:	12.35:	1.30:	5.71	17.27:	26.75:	3.18:	15.12
(33)	NONENTRY	20.15:	23.03:	15.98:	20.64	8.59:	7.36:	7.50:	9.98
(35)	MODIFIED	23.02:	24.57:	21.78:	22.76	8.76:	8.53:	9.43:	9.57
(35)	UNMODIFIED	12.94:	23.85:	3.48:	11.70	18.84:	29.64:	3.90:	10.46
(35)	MODIFIED	23.89:	27.25:	19.22:	25.03	9.12:	6.98:	9.72:	9.79
(35)	UNMODIFIED	13.25:	22.67:	4.78:	12.44	18.03:	27.95:	3.81:	12.28

Avg. is an abbreviation for AVERAGE
REL.PERCENT. is an abbreviation for RELATIVE PERCENTAGE

elaborative product aspects

(44)	CYCLOMATIC COMPLEXITY :								
(45)	SIMPPRED-NCASE VARIATION :								
(46)	TOTAL	229.5:	212.0:	250.3:	226.7	52.9:	61.0:	42.5:	55.3
(47)	#SEGS:CC>=10	4.6:	4.8:	4.8:	4.3	2.4:	1.7:	3.5:	2.1
(48)	0.5 QUANTILE POINT VALUE	3.7:	4.3:	3.5:	3.3	1.0:	0.8:	1.0:	1.0
(48)	0.5 QUANTILE TAIL AVERAGE	8.5:	10.0:	7.8:	7.7	2.6:	2.7:	2.8:	2.2
(46)	0.7 QUANTILE POINT VALUE	5.6:	7.0:	5.3:	4.6	1.5:	0.9:	1.6:	1.0

(46)		0.7 QUANTILE TAIL AVERAGE	11.4:	13.6:	10.3:	10.5	4.0:	4.3:	4.1:	3.6
(46)		0.8 QUANTILE POINT VALUE	7.7:	8.8:	7.8:	6.7	2.1:	1.2:	2.6:	1.8
(46)		0.8 QUANTILE TAIL AVERAGE	14.1:	16.4:	12.6:	13.2	5.4:	5.7:	5.5:	5.2
(46)		0.9 QUANTILE POINT VALUE	13.1:	18.0:	10.3:	11.2	6.7:	8.9:	4.7:	3.7
(46)		0.9 QUANTILE TAIL AVERAGE	19.2:	22.6:	16.0:	19.0	8.8:	9.3:	7.9:	9.3
(45)		SIMPPRED-LOGCASE VARIATION :								
(46)		TOTAL	183.8:	165.7:	219.0:	169.3	48.7:	57.2:	32.3:	41.4
(47)		#SEGS:CC>=10	2.9:	3.2:	3.7:	2.1	2.1:	1.9:	3.1:	1.2
(46)		0.5 QUANTILE POINT VALUE	3.6:	4.2:	3.5:	3.3	1.0:	0.8:	1.0:	1.0
(46)		0.5 QUANTILE TAIL AVERAGE	7.0:	8.0:	7.3:	5.8	2.0:	1.8:	2.4:	1.2
(46)		0.7 QUANTILE POINT VALUE	5.3:	6.5:	5.3:	4.3	1.6:	1.0:	1.6:	1.3
(46)		0.7 QUANTILE TAIL AVERAGE	8.9:	10.3:	9.4:	7.3	2.8:	2.6:	3.6:	1.6
(46)		0.8 QUANTILE POINT VALUE	6.7:	7.5:	7.3:	5.2	1.7:	1.5:	2.0:	1.0
(46)		0.8 QUANTILE TAIL AVERAGE	10.6:	11.9:	11.3:	8.8	3.6:	3.4:	4.6:	2.3
(46)		0.9 QUANTILE POINT VALUE	9.6:	12.3:	9.5:	7.4	3.4:	3.6:	3.3:	1.4
(46)		0.9 QUANTILE TAIL AVERAGE	13.9:	15.7:	14.8:	11.5	5.7:	5.4:	7.4:	4.0
(45)		COMPPRED-NCASE VARIATION :								
(46)		TOTAL	159.4:	140.5:	189.8:	149.6	38.7:	43.7:	23.9:	31.9
(47)		#SEGS:CC>=10	3.4:	3.5:	3.5:	3.1	1.9:	1.5:	2.8:	1.5
(46)		0.5 QUANTILE POINT VALUE	3.2:	3.3:	3.3:	2.9	0.8:	0.5:	0.8:	0.9
(46)		0.5 QUANTILE TAIL AVERAGE	7.1:	8.6:	6.4:	6.5	2.1:	2.4:	1.6:	1.7
(46)		0.7 QUANTILE POINT VALUE	4.8:	5.5:	5.0:	4.0	1.3:	1.2:	1.3:	1.0
(46)		0.7 QUANTILE TAIL AVERAGE	9.6:	11.8:	8.2:	8.8	3.2:	3.7:	2.3:	2.6
(46)		0.8 QUANTILE POINT VALUE	6.4:	7.3:	6.0:	5.9	1.6:	1.0:	1.5:	1.8
(46)		0.8 QUANTILE TAIL AVERAGE	11.8:	14.4:	9.8:	11.3	4.3:	4.9:	3.2:	4.1
(46)		0.9 QUANTILE POINT VALUE	11.4:	16.7:	8.5:	9.4	6.5:	8.8:	2.6:	3.9
(46)		0.9 QUANTILE TAIL AVERAGE	16.2:	20.5:	12.4:	15.8	7.1:	7.6:	4.5:	7.1
(45)		COMPPRED-LOGCASE VARIATION :								
(46)		TOTAL	150.6:	131.0:	182.8:	139.7	39.0:	43.7:	24.0:	30.5
(47)		#SEGS:CC>=10	2.1:	2.2:	2.8:	1.4	1.6:	1.7:	2.0:	1.0
(46)		0.5 QUANTILE POINT VALUE	3.1:	3.2:	3.3:	2.7	0.7:	0.4:	0.8:	0.8
(46)		0.5 QUANTILE TAIL AVERAGE	5.9:	6.8:	6.0:	5.0	1.4:	1.5:	1.4:	.9
(46)		0.7 QUANTILE POINT VALUE	4.6:	5.3:	4.8:	3.7	1.2:	1.0:	1.2:	0.8
(46)		0.7 QUANTILE TAIL AVERAGE	7.5:	8.7:	7.5:	6.3	1.9:	2.1:	1.8:	1.1
(46)		0.8 QUANTILE POINT VALUE	5.6:	6.5:	5.5:	4.9	1.3:	1.2:	1.4:	0.7
(46)		0.8 QUANTILE TAIL AVERAGE	8.8:	10.3:	8.8:	7.6	2.4:	2.6:	2.4:	1.6
(46)		0.9 QUANTILE POINT VALUE	8.3:	10.7:	8.0:	6.4	2.8:	2.9:	2.5:	1.3
(46)		0.9 QUANTILE TAIL AVERAGE	11.4:	13.6:	11.1:	9.7	3.6:	3.9:	3.7:	2.7
(41)		(SEGMENT,GLOBAL,SEGMENT) DATA BINDINGS :								
(42)	*	ACTUAL	419.9:	344.7:	426.7:	478.6	357.2:	142.9:	333.5:	513.2
(43)		SUBFUNCTIONAL	262.6:	249.0:	270.0:	268.0	208.0:	107.4:	217.8:	283.4
(43)		INDEPENDENT	157.3:	95.7:	156.7:	210.6	163.9:	39.0:	124.7:	243.4
(42)	*	POSSIBLE	21310:	11014:	42391:	12065	26463:	6477:	39828:	9691
(42)	*	RELATIVE PERCENTAGE	3.66:	3.83:	1.92:	5.00	3.55:	2.31:	1.49:	5.14
(49)		SOFTWARE SCIENCE QUANTITIES :								
(50)		VOCABULARY	285.9:	256.7:	307.3:	292.6	57.8:	67.4:	39.2:	59.9
(50)		LENGTH	3364.5:	3124.3:	3666.3:	3257.3	752.0:	739.6:	428.4:	962.7
(50)		ESTIMATED LENGTH	2088.3:	1840.0:	2268.3:	2146.9	520.3:	600.8:	348.7:	558.7
(50)		%DIFFERENCE(N,N^)	36.43:	41.05:	37.75:	31.34	12.62:	11.17:	8.87:	15.97

APPENDIX 1

(50)	VOLUME	27332:	25066:	30282:	26746	6684.5:	6708.9:	3747.7:	8402.1
(50,99)	INTELLIGENCE CONTENT	103.8:	101.2:	105.8:	104.4	23.0:	26.9:	11.7:	29.4
(50)	ESTIMATED BUGS	9.1:	8.5:	9.8:	8.9	2.1:	2.2:	1.2:	2.8
(51)	1ST CALCULATION METHOD :								
(50)	PROGRAM LEVEL	.00396:	.00409:	.00353:	.00421	.00101:	.00060:	.00053:	.00151
(50)	DIFFICULTY	269.3:	249.0:	288.8:	270.0	73.3:	40.0:	47.4:	110.0
(50,99)	POTENTIAL VOLUME	103.8:	101.2:	105.8:	104.4	23.0:	26.9:	11.7:	29.4
(50)	LANGUAGE LEVEL	.4192:	.4157:	.3775:	.4580	.1636:	.1368:	.0894:	.2336
(50)	EFFORT	7671.6:	6340.5:	8856.5:	7796.9	3526.4:	2234.7:	2434.4:	5001.2
(50)	ESTIMATED TIME	118.5:	98.0:	136.7:	120.4	54.5:	34.7:	37.8:	77.1
(51)	2ND CALCULATION METHOD:								
(50)	PROGRAM LEVEL	.00764:	.00734:	.00694:	.00849	.00178:	.00105:	.00106:	.00249
(50)	DIFFICULTY	136.1:	138.5:	146.7:	124.9	25.1:	21.3:	21.6:	29.2
(50)	POTENTIAL VOLUME	201.6:	179.5:	207.7:	215.4	38.6:	35.5:	17.0:	49.3
(50)	%DIFFERENCE(V*,I)	47.73:	43.65:	48.53:	50.54	10.23:	9.34:	8.83:	12.24
(50)	LANGUAGE LEVEL	1.555:	1.308:	1.450:	1.857	.587:	.289:	.292:	.845
(50)	EFFORT	3832.7:	3568.2:	4496.0:	3491.0	1433.7:	1393.6:	1125.9:	1686.8
(50)	ESTIMATED TIME	59.1:	55.0:	69.3:	53.9	22.2:	21.7:	17.4:	26.1

N.B. Values for SOFTWARE SCIENCE QUANTITIES\...\EFFORT are given in thousands of "elementary mental discriminations."
Values for SOFTWARE SCIENCE QUANTITIES\...\ESTIMATED TIME are given in hours.

APPENDIX 1

References

[Baker 72] F.T. Baker. Chief programmer team management of production programming. IBM Systems Journal, vol. 11, no. 1 (1972), pp. 56-73.

[Baker 75] F.T. Baker. Structured programming in a production programming environment. IEEE Transactions on Software Engineering, vol. 1, no. 2 (June 1975), pp. 241-252.

[Basili & Baker 77] V.R. Basili and F.T. Baker. Tutorial of Structured Programming. IEEE Catalog No. 75CH1049-6, Tutorial from the Eleventh IEEE Computer Society Conference (COMPCON 75 Fall), revised 1977.

[Basili & Reiter 78] V.R. Basili and R.W. Reiter, Jr. Investigating software development approaches. Technical Report TR-688, Department of Computer Science, University of Maryland, August, 1978.

[Basili & Reiter 79a] V.R. Basili and R.W. Reiter, Jr. Evaluating automatable measures of software development. Proceedings of the IEEE/Poly Workshop on Quantitative Software Models for Reliability, Complexity, and Cost (October 1979), Kiameshia Lake, New York.

[Basili & Reiter 79b] V.R. Basili and R.W. Reiter, Jr. An investigation of human factors in software development. Computer Magazine, vol. 12, no. 12 (December 1979), pp. 21-38.

[Basili & Reiter 80] V.R. Basili and R.W. Reiter, Jr. A controlled experiment quantitatively comparing software development approaches. to be published in IEEE Transactions on Software Engineering, 1980.

[Basili & Turner 75] V.R. Basili and A.J. Turner. Iterative enhancement: a practical technique for software development. IEEE Transactions on Software Engineering, vol. 1, no. 4 (December 1975), pp.

393-396.

[Basili & Turner 76] V.R. Basili and A.J. Turner. SIMPL-I, A Structured Programming Language. Geneva, Illinois: Paladin House, 1976.

[Basili & Zelkowitz 78] V.R. Basili and M.V. Zelkowitz. Analyzing medium-scale software development. IEEE Catalog No. 78CH1317-7C, Proceedings of the Third International Conference on Software Engineering (May 1978), Atlanta, Georgia, pp. 116-123.

[Basili & Zelkowitz 79] V.R. Basili and M.V. Zelkowitz. Measuring software development characteristics in the local environment. Computers & Structures, vol. 10 (1979), pp. 39-43.

[Basili et al. 77] V.R. Basili, M.V. Zelkowitz, F.E. McGarry, R.W. Reiter, Jr., W.F. Truszkowski, and D.L. Weiss. The software engineering laboratory. Technical Report TR-535, Department of Computer Science, University of Maryland, May, 1977.

[Belady & Lehman 76] L.A. Belady and M.M. Lehman. A model of large program development. IBM Systems Journal, vol. 15, no. 3 (1976), pp. 225-251.

[Berge 73] C. Berge. Graphs and Hypergraphs. Amsterdam, The Netherlands: North-Holland, 1973.

[Brooks 75] F.P. Brooks, Jr. The Mythical Man-Month. Reading, Massachusetts: Addison-Wesley, 1975.

[Campbell & Stanley 63] D.T. Campbell and J.C. Stanley. Experimental and Quasi-Experimental Designs for Research. Reprinted from Handbook of Research on Teaching, edited by N.L. Gage. Chicago, Illinois: Rand McNally, 1963.

[Conover 71] W.J. Conover. Practical Nonparametric Statistics. New York, New York: John Wiley & Sons, 1971.

[Curtis et al. 79] B. Curtis, S.B. Sheppard, P. Milliman, M.A. Borst, and T. Love. Measuring the psychological

complexity of software maintenance tasks with Halstead and McCabe metrics. IEEE Transactions on Software Engineering, vol. 5, no. 2 (March 1979), pp. 95-104.

[Dahl, Dijkstra & Hoare 72] O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. Structured Programming. New York, New York: Academic Press, 1972.

[Daley 77] E.B. Daley. Management of software development. IEEE Transactions on Software Engineering, vol. 3, no. 3 (May 1977), pp. 229-242.

[Dunsmore 78] H.E. Dunsmore. The influence of programming factors on programming complexity. Ph.D. Dissertation. Department of Computer Science, University of Maryland, July, 1978. available as Technical Report TR-679.

[Dunsmore & Gannon 77] H.E. Dunsmore and J.D. Gannon. Experimental investigation of programming complexity. Proceedings of the ACM/NBS Sixteenth Annual Technical Symposium: Systems and Software (June 1977), Washington, D.C., pp. 117-125.

[Elshoff 76a] J.L. Elshoff. Measuring commercial PL/1 programs using Halstead's criteria. ACM SIGPLAN Notices, vol. 11, no. 5 (May 1976), pp. 38-46.

[Elshoff 76b] J.L. Elshoff. An analysis of some commercial PL/1 programs. IEEE Transactions on Software engineering, vol. 2, no. 2 (June 1976), pp. 113-120.

[Fagan 76] M.E. Fagan. Design and code inspections to reduce errors in program development. IBM Systems Journal, vol. 15, no. 3 (1976), pp. 182-211.

[Fitzsimmons & Love 78] A. Fitzsimmons and T. Love. A review and evaluation of software science. Computing Surveys, vol. 10, no. 1 (March 1978), pp. 3-18.

[Gannon 75] J.D. Gannon. Language design to enhance programming reliability. Ph.D. Thesis. Department of Computer Science, University of Toronto, January, 1975. available as Technical Report CSRG-47.

[Gannon 77] J.D. Gannon. An experimental evaluation of

data type conventions. Communications of the ACM, vol. 20, no. 8 (August 1977), pp. 564-595.

[Gannon & Horning 75] J.D. Gannon and J.J. Horning. Language design for programming reliability. IEEE Transactions on Software Engineering, vol. 1, no. 2 (June 1975), pp. 179-191.

[Gilb 77] T. Gilb. Software Metrics. Cambridge, Massachusetts: Winthrop, 1977.

[Gordon 79] R.D. Gordon. A qualitative justification for a measure of program clarity. IEEE Transactions on Software Engineering, vol. 5, no. 2 (March 1979), pp. 121-128.

[Green 77] T.R.G. Green. Conditional program statements and their comprehensibility to professional programmers. Journal of Occupational Psychology, vol. 50 (1977), pp. 93-109.

[Halstead 77] M. Halstead. Elements of Software Science. New York, New York: Elsevier, 1977.

[Hughes 79] J.W. Hughes. A formalization and explication of the Michael Jackson method of program design. Software--Practice and Experience, vol. 9, no. 3 (March 1979), pp. 191-202.

[Jackson 75] M.A. Jackson. Principles of Program Design. New York, New York: Academic Press, 1975.

[Kirk 68] R.E. Kirk. Experimental Design: Procedures for the Behavioral Sciences. Belmont, California: Wadsworth, 1968.

[Knuth 71] D.E. Knuth. An empirical study of FORTRAN programs. Software--Practice and Experience, vol. 1, no. 2 (April-June 1971), pp. 105-133.

[Linger, Mills & Witt 79] R.C. Linger, H.D. Mills, and B.I. Witt. Structured Programming: Theory and Practice. Reading, Massachusetts: Addison-Wesley, 1979.

[Love & Bowman 76] L.T. Love and A.B. Bowman. An independent test of the theory of software physics.

ACM SIGPLAN Notices, vol. 11, no. 11 (November 1976), pp. 42-49.

[McCabe 76] T.J. McCabe. A complexity measure. IEEE Transactions on Software Engineering, vol. 2, no. 4 (December 1976), pp. 308-320.

[Mills 72] H.D. Mills. Mathematical foundations for structured programming. FSC 72-6012, IBM Corporation, Gaithersburg, Maryland, February, 1972.

[Mills 73] H.D. Mills. The complexity of programs. in Program Test Methods, edited by W.C. Hetzel, pp. 225-238. Englewood Cliffs, New Jersey: Prentice-Hall, 1973.

[Mills 76] H.D. Mills. Software development. IEEE Transactions on Software Engineering, vol. 2, no. 4 (December 1976), pp. 265-273.

[Myers 75] G.J. Myers. Reliable Software through Composite Design. New York, New York: Petrocelli/Charter, 1975.

[Myers 77] G.J. Myers. An extension to the cyclomatic measure of program complexity. ACM SIGPLAN Notices, vol. 12, no. 10 (October 1977), pp. 61-64.

[Myers 78] G.J. Myers. A controlled experiment in program testing and code walkthroughs/inspections. Communications of the ACM, vol. 21, no. 9 (September 1978), pp. 760-768.

[Nemenyi et al. 77] P. Nemenyi, S.K. Dixon, N.B. White, Jr., and M.L. Headstrom. Statistics from Scratch. San Francisco, California: Holden-Day, 1977.

[Oldehoeft 77] R.R. Oldehoeft. A contrast between language level measures. IEEE Transactions on Software Engineering, vol. 3, no. 6 (November 1977), pp. 476-478.

[Ostle & Mensing 75] B. Ostle and R.W. Mensing. Statistics in Research, Third Edition. Ames, Iowa: Iowa State University Press, 1975.

[Putnam 78] L.H. Putnam. A general empirical solution to

the macro software sizing and estimation problem. IEEE Transactions on Software Engineering, vol. 4, no. 4 (July 1978), pp. 301-316.

[Reiter 79] R.W. Reiter, Jr. Empirical investigation of computer program development approaches and computer programming metrics. Ph.D. Dissertation. Department of Computer Science, University of Maryland, December, 1979.

[Sheppard et al. 79] S.B. Sheppard, B. Curtis, P. Milliman, and T. Love. Modern coding practices and programmer performance. Computer Magazine, vol. 12, no. 12 (December 1979), pp. 41-49.

[Shneiderman 76] B. Shneiderman. Exploratory experiments in programmer behavior. Internation Journal of Computer and Information Sciences, vol. 5, no. 2 (June 1976), pp. 123-143.

[Shneiderman 80] B. Shneiderman. Software Psychology: Human Factors in Computer and Information Systems. Cambridge, Massachusetts: Winthrop, 1980.

[Shneiderman et al. 77] B. Shneiderman, R. Mayer, D. McKay, and P. Heller. Experimental investigations of the utility of detailed flowcharts in programming. Communications of the ACM, vol. 20, no. 6 (June 1977), pp. 373-381.

[Shooman 78] M. Shooman. Private communication, in conjunction with a colloquium (on Software Reliability Models) given at the Department of Computer Science, university of Maryland, October, 1978.

[Siegel 56] S. Siegel. Nonparametric Statistics: for the Behavioral Sciences. New York, New York: McGraw-Hill, 1956.

[Sime, Green & Guest 73] M.E. Sime, T.R.G. Green, and D.J. Guest. Psychological evaluation of two conditional constructs used in computer languages. International Journal of Man-Machine Studies, vol. 5, no. 1 (January

1973), pp. 105-113.

[Simon 69] H.A. Simon. The architecture of complexity. in The Sciences of the Artificial, pp. 84-118. Cambridge, Massachusetts: MIT Press, 1969.

[Stevens 46] S.S. Stevens. On the theory of scales of measurement. Science, vol. 103 (1946), pp. 677-680.

[Stevens, Myers & Constantine 74] W.P. Stevens, G.J. Myers, and L.L. Constantine. Structured design. IBM Systems Journal, vol. 13, no. 2 (1974), pp. 115-139.

[Tukey 69] J.W. Tukey. Analyzing data: sanctification or detective work? American Psychologist, vol. 24, no. 2 (February 1969), pp. 83-91.

[Turner 76] A.J. Turner, Jr. Iterative enhancement: a practical technique for software development. Ph.D. Dissertation. Department of Computer Science, University of Maryland, May, 1976.

[Walston & Felix 77] C.E. Walston and C.P. Felix. A method of programming measurement and estimation. IBM Systems Journal, vol. 16, no. 1 (1977), pp. 54-73.

[Weinberg 71] G.M. Weinberg. The Psychology of Computer Programming. New York, New York: Van Nostrand Reinhold, 1971.

[Weissman 74a] L.M. Weissman. Psychological complexity of computer programs: an experimental methodology. ACM SIGPLAN Notices, vol. 9, no. 6 (June 1974), pp. 25-36.

[Weissman 74b] L.M. Weissman. A methodology for studying the psychological complexity of computer programs. Ph.D. Thesis. Department of Computer Science, University of Toronto, August, 1974. available as Technical Report CSRG-37.

[Wirth 71] N. Wirth. Program development by stepwise refinement. Communications of the ACM, vol. 14, no. 4 (April 1971), pp. 221-227.

CURRICULUM VITAE

Name: Robert William Reiter, Jr.

Permanent address: 900 Jamieson Road,
Lutherville, Maryland 21093.

Degree and date to be conferred: Ph.D., December, 1979.

Date of birth: June 7, 1950.

Place of birth: Baltimore, Maryland.

Secondary education: Loyola High School, Towson, Maryland,
June, 1968.

Collegiate Institutions	Dates	Degree	Date of Degree
Massachusetts Institute of Technology	1968-72	S.B.	June, 1972.
University of Maryland	1972-76	M.S.	May, 1976.
University of Maryland	1976-79	Ph.D.	December, 1979.

Major: Computer Science.

Professional publications:

V.R. Basili, M.V. Zelkowitz, F.E. McGarry, R.W. Reiter, Jr., W.F. Truszkowski, and D.L. Weiss.
The software engineering laboratory.
Technical Report TR-535, Department of Computer Science, University of Maryland, May, 1977.

V.R. Basili and R.W. Reiter, Jr.
Investigating software development approaches.
Technical Report TR-688, Department of Computer Science, University of Maryland, August, 1978.

V.R. Basili and R.W. Reiter, Jr.
Evaluating automatable measures of software development.
Proceedings of the IEEE/Poly Workshop on Quantitative Software Models for Reliability, Complexity, and Cost (IEEE Catalog No. #), Kiameshia Lake, NY (October 1979), pp. #-#.

V.R. Basili and R.W. Reiter, Jr.
An investigation of human factors in software development.
Computer Magazine, vol. 12, no. 12 (December 1979), pp. 21-38.

V.R. Basili and R.W. Reiter, Jr.
A controlled experiment quantitatively comparing software development approaches.
(to be published in IEEE Transactions on Software Engineering, 1980)

Professional positions held:

1970-1973 (Summers only) -- Programmer/Analyst
Information Systems Department, Baltimore Gas & Electric Company, Baltimore, Maryland 21203.

1972-1976 -- Graduate Teaching Assistant
Department of Computer Science, University of Maryland, College Park, Maryland 20742.

1976-1979 -- Graduate Research Assistant
Department of Computer Science, University of Maryland, College Park, Maryland 20742.

1979- -- Faculty Research Assistant
Department of Computer Science, University of Maryland, College Park, Maryland 20742.


DAT
ILM